SOFTWARE THEFT DETECTION THROUGH PROGRAM IDENTIFICATION

by

Ginger Marie Myles

Copyright © Ginger Marie Myles 2006

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

2006
THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation
prepared by Ginger Marie Myles
entitled Software Theft Detection Through Program Identification
and recommend that it be accepted as fulfilling the dissertation requirement for the
Degree of Doctor of Philosophy

__________________________ Date: 3/31/06
Christian S. Collberg

__________________________ Date: 3/31/06
Saumya Debray

__________________________ Date: 3/31/06
Stephen Kobourov

__________________________ Date: 3/31/06
John H. Hartman

Final approval and acceptance of this dissertation is contingent upon the candidate’s
submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and
recommend that it be accepted as fulfilling the dissertation requirement.

__________________________ Date: 3/31/06
Dissertation Director: Christian S. Collberg
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: Ginger M. Myles
ACKNOWLEDGEMENTS

I owe a very special thanks to my advisor, Professor Christian Collberg. Several years ago he gave me a chance when I was struggling to find direction. In the years that followed he has been an excellent mentor and friend. He coached me through my first conference publication and talk, taught me to shorten my sentences, and how to make better use of the comma. He has shown me the joy in research and in inventing new ideas. He has been exactly what an advisor should be. I am truly grateful for everything that he has taught me.

I also want to thank the SANDMARK team. I learned an immeasurable amount from working on this team and without everyone’s effort this dissertation would be much shorter. Specifically, I want to thank Andy, Jas, Kelly, and Mike who I worked so closely with over several years. We had a great run, its a shame it had to end.

Thanks also go to my team at the IBM Almaden Research Center who have given me a wonderful opportunity to continue exploring the topic I love. The entire team has been extremely supportive during the completion of my dissertation. I especially want to thank Stefan who supported my work through weekly conversations and valuable comments on early drafts.

A variety of people in Tucson deserve special note. First, Sonia has been an invaluable ally in the long battle of obtaining my PhD. She was always there when I had a question, needed a few words of encouragement, or just wanted to talk. My Tucson buddies, Judd, Justine, and Wendy. What would I have done without long bike rides and fried tofu with spiced salt. And finally Suzanne, who provided me with a family when mine was so far away. Being assigned as her TA in my second semester as a grad student was definitely a blessing. From that experience I learned many things about being a great educator.

I also wish to give my family my deepest appreciation. I know that my parents made many sacrifices to provide me with the education that laid the foundation for me to successfully reach my goals. Both my parents and grandparents provided me with encouragement and support as I explored many avenues in my attempt to discover who I was and what I wanted to do. I am eternally grateful to my grandfather who showed me the joy in hard work, learning, teaching, numbers, and computers.

Last, but definitely not least, I want to thank my partner Stephanie. I can only imagine how boring it was for her to proof-read my papers, but she enthusiastically read every single one, and some multiple times. Without her sacrifices and endless support reaching this goal would have been an impossible battle.
In loving memory of my grandfather, Meredith Marion Myles, Sr.
TABLE OF CONTENTS

LIST OF FIGURES ................................. 12
LIST OF TABLES ................................. 15
ABSTRACT ........................................ 17

CHAPTER 1 INTRODUCTION ....................... 18
  1.1 Contributions .............................. 19
  1.2 Organization .............................. 21

I  BACKGROUND .................................... 26

CHAPTER 2 SOFTWARE PIRACY ................... 27
  2.1 Deterrent Solutions ....................... 31
      2.1.1 United States Copyright Law ....... 31
      2.1.2 United States Patent Law ......... 33
      2.1.3 United States Trademark Law ..... 34
      2.1.4 United States Trade Secret Law ... 34
  2.2 Preventive Solutions ...................... 35
      2.2.1 Audits ................................ 35
      2.2.2 Hardware-Based Techniques ........ 35
      2.2.3 Software-Based Techniques ........ 38
  2.3 Summary .................................... 43

CHAPTER 3 TERMINOLOGY ....................... 44
  3.1 Control Flow Graph ....................... 44
  3.2 Graph Coloring Register Allocator ....... 45
  3.3 Dominator Set ............................ 47
  3.4 Opaque Predicates ....................... 48
  3.5 Branch Function ......................... 49

II  SANDMARK ...................................... 52

CHAPTER 4 THE SANDMARK TOOL ................. 53
  4.1 System Architecture ...................... 53
TABLE OF CONTENTS – Continued

4.2 Support Code ................................................................. 58
4.3 Protection Modules ......................................................... 59
4.4 Evaluation Tools .............................................................. 62
  4.4.1 Software Statistics ...................................................... 62
  4.4.2 View Pane ............................................................... 64
  4.4.3 Java Bytecode Comparison Tool ................................. 65
  4.4.4 Optimization ........................................................... 68
4.5 Discussion ................................................................. 69

III SOFTWARE WATERMARKING ......................... 70

CHAPTER 5 OVERVIEW .......................................................... 71
  5.1 Applications of Software Watermarking ......................... 71
  5.2 Definition ............................................................... 73
  5.3 Software Watermarking Classification ............................ 75
    5.3.1 Static or Dynamic Code Properties .......................... 75
    5.3.2 Recognition Type ................................................ 78
    5.3.3 Embedding Technique .......................................... 78
    5.3.4 Embedded Mark Type .......................................... 79
  5.4 Software Watermark Evaluation ..................................... 80
    5.4.1 Threat Model .................................................... 80
    5.4.2 Evaluation Properties ........................................ 81
    5.4.3 Evaluation Framework ........................................ 83

CHAPTER 6 NAÏVE WATERMARKING ALGORITHMS .......... 87
  6.1 Constant String ........................................................ 87
  6.2 Identifier Encoding ................................................... 89
  6.3 Switch Encoding ....................................................... 90
  6.4 Discussion ............................................................. 92

CHAPTER 7 PUBLISHED WATERMARKING TECHNIQUES ...... 93
  7.1 Davidson-Myhrvold Algorithm ...................................... 93
    7.1.1 MIPS Version ................................................... 94
    7.1.2 Java Version ................................................... 95
  7.2 Moskowitz-Cooperman Algorithm ................................ 105
    7.2.1 Embedding ..................................................... 107
    7.2.2 Recognition .................................................... 108
    7.2.3 Evaluation ..................................................... 110
  7.3 Graph Coloring-Based Watermarking .............................. 113
### TABLE OF CONTENTS – *Continued*

7.3.1 Embedding .............................................. 114  
7.3.2 Recognition ......................................... 115  
7.3.3 Evaluation ........................................... 116  

7.4 Spread Spectrum Watermarking .................................. 120  
7.4.1 Embedding .............................................. 120  
7.4.2 Recognition ........................................... 121  
7.4.3 Evaluation ............................................. 123  

7.5 Dummy Method ............................................. 129  
7.5.1 Embedding .............................................. 129  
7.5.2 Recognition ........................................... 131  
7.5.3 Evaluation ............................................. 132  

7.6 Graph-Theoretic Watermarking ................................... 138  
7.6.1 Embedding .............................................. 138  
7.6.2 Recognition ........................................... 141  
7.6.3 Evaluation ............................................. 142  

7.7 Aboit Algorithm ............................................. 145  
7.7.1 Embedding .............................................. 147  
7.7.2 Recognition ........................................... 149  
7.7.3 Evaluation ............................................. 150  

7.8 Collberg-Thompsonson Algorithm .................................. 157  
7.8.1 Embedding .............................................. 157  
7.8.2 Recognition ........................................... 159  
7.8.3 Evaluation ............................................. 159  

7.9 Path-Based Watermarking ....................................... 163  
7.9.1 Java Version ........................................... 163  
7.9.2 Native Code Version .................................... 169  
7.9.3 Evaluation ............................................. 171  

7.10 Thread-Based Watermarking ..................................... 174  
7.10.1 Embedding .............................................. 174  
7.10.2 Recognition ........................................... 175  
7.10.3 Evaluation ............................................. 176  

7.11 Abstract Interpretation Watermarking .................................. 179  
7.11.1 Embedding .............................................. 179  
7.11.2 Recognition ........................................... 183  
7.11.3 Evaluation ............................................. 183  

7.12 Discussion .................................................. 188
TABLE OF CONTENTS – Continued

CHAPTER 8 ALGORITHM EXTENSIONS .................... 191
   8.1 Graph Coloring-Based Watermarking ................. 191
      8.1.1 Algorithmic Shortcomings ................. 191
      8.1.2 Algorithmic Improvements ................. 194
      8.1.3 Evaluation ................................ 197
   8.2 Arboit Algorithm .................................. 200
      8.2.1 Evaluation ................................ 204
   8.3 Discussion ....................................... 205

CHAPTER 9 BRANCH-BASED SOFTWARE WATERMARKING 208
   9.1 The Branch-Based Algorithm ....................... 209
   9.2 Fingerprint Branch Function ....................... 211
   9.3 The Branch-Based Algorithm – Native Code Implementation .... 215
      9.3.1 Annotation ................................ 217
      9.3.2 Tracing .................................... 217
      9.3.3 Embedding ................................ 219
      9.3.4 Extraction ................................ 226
      9.3.5 Registration-Based Customization .......... 226
      9.3.6 Evaluation ................................ 229
      9.3.7 Improving Resilience ....................... 237
   9.4 The Branch-Based Algorithm – Java Bytecode Implementation .... 238
      9.4.1 Annotation ................................ 240
      9.4.2 Tracing .................................... 240
      9.4.3 Embedding ................................ 241
      9.4.4 Extraction ................................ 251
      9.4.5 Evaluation ................................ 252
   9.5 Summary .......................................... 255

IV SOFTWARE BIRTHMARKING .......................... 257

CHAPTER 10 OVERVIEW .................................. 258
   10.1 Applications of Software Birthmarking ............. 258
   10.2 Definition ...................................... 259
      10.2.1 Evaluating Software Birthmarks ............ 262
   10.3 Evaluation Framework ............................. 262
   10.4 Related Work ................................... 264
   10.5 Birthmarks and Watermarks ....................... 267
TABLE OF CONTENTS – Continued

CHAPTER 11 PUBLISHED BIRTHMARKING TECHNIQUES . 268
11.1 TaNaMM Birthmark .................................................. 268
   11.1.1 Similarity of TaNaMM Birthmarks ......................... 271
   11.1.2 Evaluation ....................................................... 273
11.2 K-Gram Based Birthmark ........................................... 277
   11.2.1 Similarity of K-Gram Birthmarks ......................... 281
   11.2.2 Evaluation ....................................................... 284
11.3 K-Gram Birthmark vs. TaNaMM Birthmark .................... 288

CHAPTER 12 WHOLE PROGRAM PATH BIRTHMARKING . 291
12.1 Whole Program Paths ............................................... 291
   12.1.1 Similarity of WPP Birthmarks ............................ 294
   12.1.2 Example ......................................................... 295
   12.1.3 Computing the Maximum Common Subgraph ............. 297
12.2 Evaluation .......................................................... 302
   12.2.1 Credibility ...................................................... 302
   12.2.2 Resistance to Transformation ............................. 303
   12.2.3 Technique Comparison ................................... 305
   12.2.4 Birthmarks and Watermarks ............................... 307
12.3 Summary ............................................................. 310

CHAPTER 13 CONCLUSION ................................................. 311
13.1 Future Work ........................................................ 313

APPENDIX A ASSEMBLY LEVEL PROGRAMMING: JAVA AND
 x86 ................................................................. 315
 A.1 Java Bytecode and the Java Virtual Machine ............... 315
   A.1.1 The Class File Format .................................. 316
   A.1.2 The JVM ......................................................... 317
   A.1.3 Java Bytecode ................................................ 320
 A.2 x86 Assembly Language Programming .......................... 320

APPENDIX B Description of SandMark Obfuscations ............ 324

APPENDIX C SPECint-2000 Benchmark Suite Configuration File 328

APPENDIX D Factorial and Fibonacci Programs .................. 330

APPENDIX E Birthmark Similarity Results ........................ 332
TABLE OF CONTENTS – Continued

APPENDIX F Complete Publications List .................... 340
## LIST OF FIGURES

| FIGURE 2.1 | Illegal software sales on Internet auction sites. | 29 |
| FIGURE 2.2 | Simple obfuscating transformation. | 40 |
| FIGURE 3.1 | Method and corresponding control flow graph. | 46 |
| FIGURE 3.2 | Sample code and corresponding interference graph. | 47 |
| FIGURE 3.3 | Branch function transformation. | 50 |
| FIGURE 4.1 | The SandMark graphical user interface. | 54 |
| FIGURE 4.2 | The sandmark.Algorithm class. | 56 |
| FIGURE 4.3 | Sample obfuscation. | 57 |
| FIGURE 4.4 | Obfuscation customization. | 60 |
| FIGURE 4.5 | Obfuscation executive configuration. | 61 |
| FIGURE 4.6 | SandMark View pane illustration. | 65 |
| FIGURE 4.7 | Class level static code statistics. | 66 |
| FIGURE 4.8 | Viewing of bytecode instructions. | 67 |
| FIGURE 4.9 | SandMark Diff tool. | 68 |
| FIGURE 5.1 | Adobe Acrobat easter egg watermark. | 77 |
| FIGURE 6.1 | Identifier Encoding watermark embedding. | 89 |
| FIGURE 7.1 | Classification of published watermarking algorithms. | 94 |
| FIGURE 7.2 | DM watermark transformation example. | 98 |
| FIGURE 7.3 | Maximum size watermark embedded using DM algorithm. | 101 |
| FIGURE 7.4 | Size impact incurred using the DM algorithm. | 103 |
| FIGURE 7.5 | Moskowitz-Cooperman watermarking transformation. | 109 |
| FIGURE 7.6 | Size impact incurred using Moskowitz-Cooperman. | 111 |
| FIGURE 7.7 | Illustration of Graph Coloring-Based recognition cases. | 116 |
| FIGURE 7.8 | Collusive attack on the Graph Coloring-Based algorithm. | 119 |
| FIGURE 7.9 | Example Spread Spectrum code book entries. | 122 |
| FIGURE 7.10 | Size impact incurred using Spread Spectrum. | 125 |
| FIGURE 7.11 | Collusive attack on the Spread Spectrum algorithm. | 129 |
| FIGURE 7.12 | Illustration of Dummy Method algorithm. | 134 |
| FIGURE 7.13 | Size impact incurred using the Dummy Method algorithm. | 135 |
| FIGURE 7.14 | Illustration of the GTW watermarking transformation. | 141 |
| FIGURE 7.15 | Example instruction sequence generated by GTW<sub>SM</sub>. | 143 |
| FIGURE 7.16 | Size impact incurred using the GTW<sub>SM</sub> algorithm. | 144 |
LIST OF FIGURES – Continued

FIGURE 7.17 Size impact incurred using A1 and A2. . . . . . . . . 153
FIGURE 7.18 Size impact using A1 and A2 with constant encoding. . . 154
FIGURE 7.19 Size impact incurred using the CT algorithm. . . . . . 161
FIGURE 7.20 Size impact incurred using the Java Path-Based algorithm. 167
FIGURE 7.21 Illustration of Path-Based watermark for native code. . . 171
FIGURE 7.22 Illustration of Thread-Based watermarking. . . . . 175
FIGURE 7.23 Illustration of Abstract Interpretation watermarking. . . 184
FIGURE 7.24 Collusive attack on the Abstract Interpretation algorithm. . 188

FIGURE 8.1 Graph Coloring-Based algorithm recognition failure. . . . 193
FIGURE 8.2 Graph Coloring-Based algorithm recognition failure. . . . 194
FIGURE 8.3 Graph Coloring-Based,SM watermarking transformation. . 197
FIGURE 8.4 Illustration of the Dynamic Arboit algorithm. . . . . . 203

FIGURE 9.1 Overview of the Branch-Based algorithm. . . . . . . . 210
FIGURE 9.2 Example of Branch-Based code transformation. . . . . 212
FIGURE 9.3 Single execution of a fingerprint branch function. . . . . 213
FIGURE 9.4 Illustration of a single step in the key evolution process. . 214
FIGURE 9.5 Overview of Branch-Based watermarking. . . . . . . 215
FIGURE 9.6 Branch-Based annotation transformation. . . . . . . 218
FIGURE 9.7 Example FBF assembly instructions. . . . . . . . 222
FIGURE 9.8 Illustration of possible irregular key evolution. . . . . 224
FIGURE 9.9 Illustration of fingerprint generation. . . . . . . . 227
FIGURE 9.10 Installation and registration of watermarked software. . 228
FIGURE 9.11 Change in control flow interpretation due to watermarking. 234
FIGURE 9.12 Change in control flow interpretation due to ICBF. . . . . 239
FIGURE 9.13 Incorporation of an additional level of indirection. . . . 240
FIGURE 9.14 Example of Java Branch-Based transformation. . . . . 242
FIGURE 9.15 List of exceptions used in Java implementation. . . . . 245
FIGURE 9.16 Illustration of the exception table entries. . . . . . . 249
FIGURE 9.17 Illustration of Java Branch-Based code transformation. . 250

FIGURE 11.1 A class from the SandMark tool. . . . . . . . 269
FIGURE 11.2 Bytecode method and associated K-Gram birthmark. . 279
FIGURE 11.3 Java classfile and associated K-Gram birthmark. . . . 280
FIGURE 11.4 Frequency of bytecode sequences of size 1 ≤ k ≤ 4. . . 281
FIGURE 11.5 Frequency of bytecode sequences of size 5 ≤ k ≤ 7. . . 282
FIGURE 11.6 K-Gram resistance results where k = 2, 3, 4. . . . . 287

FIGURE 12.1 Illustration of the WPP birthmark construction. . . . . 293
LIST OF FIGURES – Continued

FIGURE 12.2 WPP birthmark of a factorial program. ............... 296
FIGURE 12.3 WPP birthmark of an obfuscated factorial program. .... 297
FIGURE 12.4 Illustration of the WPP similarity measure. ............ 298
FIGURE 12.5 Illustration of the error in the heuristic MCS technique. . 299

FIGURE A.1 Java method and corresponding bytecode. ............. 321
FIGURE A.2 x86 stack. ............................................. 322
FIGURE A.3 Example assembly instructions. ........................ 323
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 3.1</td>
<td>Number-theoretically true opaque predicates.</td>
<td>49</td>
</tr>
<tr>
<td>TABLE 5.1</td>
<td>Descriptions of watermark evaluation applications.</td>
<td>84</td>
</tr>
<tr>
<td>TABLE 5.2</td>
<td>Characteristics of watermark evaluation applications.</td>
<td>85</td>
</tr>
<tr>
<td>TABLE 7.1</td>
<td>Ratio of goto instructions using DM algorithm.</td>
<td>102</td>
</tr>
<tr>
<td>TABLE 7.2</td>
<td>DM CaffeineMark scores.</td>
<td>104</td>
</tr>
<tr>
<td>TABLE 7.3</td>
<td>DM distortive attack results.</td>
<td>106</td>
</tr>
<tr>
<td>TABLE 7.4</td>
<td>Moskowitz-Cooperman CaffeineMark scores.</td>
<td>112</td>
</tr>
<tr>
<td>TABLE 7.5</td>
<td>Spread Spectrum CaffeineMark scores.</td>
<td>126</td>
</tr>
<tr>
<td>TABLE 7.6</td>
<td>Spread Spectrum distortive attack results.</td>
<td>127</td>
</tr>
<tr>
<td>TABLE 7.7</td>
<td>Dummy Method CaffeineMark scores.</td>
<td>136</td>
</tr>
<tr>
<td>TABLE 7.8</td>
<td>Dummy Method distortive attack results.</td>
<td>137</td>
</tr>
<tr>
<td>TABLE 7.9</td>
<td>GTW$_{SM}$ CaffeineMark scores.</td>
<td>145</td>
</tr>
<tr>
<td>TABLE 7.10</td>
<td>GTW$_{SM}$ distortive attack results.</td>
<td>146</td>
</tr>
<tr>
<td>TABLE 7.11</td>
<td>Sample opaque predicate library.</td>
<td>147</td>
</tr>
<tr>
<td>TABLE 7.12</td>
<td>Maximum characters embedded using ArboIt.</td>
<td>151</td>
</tr>
<tr>
<td>TABLE 7.13</td>
<td>Static statistics of TTT and Java Grande.</td>
<td>152</td>
</tr>
<tr>
<td>TABLE 7.14</td>
<td>A1 and A2 CaffeineMark scores.</td>
<td>155</td>
</tr>
<tr>
<td>TABLE 7.15</td>
<td>ArboIt distortive attack results.</td>
<td>156</td>
</tr>
<tr>
<td>TABLE 7.16</td>
<td>Path-Based CaffeineMark scores.</td>
<td>168</td>
</tr>
<tr>
<td>TABLE 7.17</td>
<td>Hypothetical size impact using Thread-Based algorithm.</td>
<td>178</td>
</tr>
<tr>
<td>TABLE 7.18</td>
<td>Comparison of nine watermarking algorithms.</td>
<td>189</td>
</tr>
<tr>
<td>TABLE 8.1</td>
<td>Graph Coloring-Based$_{SM}$ data-rate.</td>
<td>199</td>
</tr>
<tr>
<td>TABLE 8.2</td>
<td>Graph Coloring-Based$_{SM}$ distortive attack results.</td>
<td>201</td>
</tr>
<tr>
<td>TABLE 8.3</td>
<td>Dynamic ArboIt distortive attack results.</td>
<td>206</td>
</tr>
<tr>
<td>TABLE 9.1</td>
<td>Execution impact using Branch-Based native code.</td>
<td>231</td>
</tr>
<tr>
<td>TABLE 9.2</td>
<td>Size impact using Branch-Based native code.</td>
<td>232</td>
</tr>
<tr>
<td>TABLE 9.3</td>
<td>Optimal size impact using Branch-Based native code.</td>
<td>233</td>
</tr>
<tr>
<td>TABLE 9.4</td>
<td>Total versus usable branches in SPECint-2000 applications.</td>
<td>235</td>
</tr>
<tr>
<td>TABLE 9.5</td>
<td>Total versus usable branches in Java test applications.</td>
<td>254</td>
</tr>
<tr>
<td>TABLE 10.1</td>
<td>Characteristics of birthmark evaluation applications.</td>
<td>263</td>
</tr>
<tr>
<td>TABLE 10.2</td>
<td>Descriptions of birthmark evaluation applications.</td>
<td>264</td>
</tr>
<tr>
<td>TABLE 10.3</td>
<td>Six sets of SANDMARK obfuscations.</td>
<td>265</td>
</tr>
</tbody>
</table>
**LIST OF TABLES – Continued**

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 11.1</td>
<td>TaNaMM results using different similarity measures.</td>
<td>274</td>
</tr>
<tr>
<td>TABLE 11.2</td>
<td>TaNaMM independence experiment results.</td>
<td>275</td>
</tr>
<tr>
<td>TABLE 11.3</td>
<td>TaNaMM false positive experiment results.</td>
<td>276</td>
</tr>
<tr>
<td>TABLE 11.4</td>
<td>TaNaMM resistance experiment results.</td>
<td>277</td>
</tr>
<tr>
<td>TABLE 11.5</td>
<td>Unique and total k-grams for $1 \leq k \leq 7$.</td>
<td>283</td>
</tr>
<tr>
<td>TABLE 11.6</td>
<td>K-Gram independence experiment results.</td>
<td>285</td>
</tr>
<tr>
<td>TABLE 11.7</td>
<td>K-Gram false positive experiment results.</td>
<td>286</td>
</tr>
<tr>
<td>TABLE 11.8</td>
<td>TaNaMM and K-Gram resistance results.</td>
<td>289</td>
</tr>
<tr>
<td>TABLE 11.9</td>
<td>TaNaMM and K-Gram false positive results.</td>
<td>290</td>
</tr>
<tr>
<td>TABLE 12.1</td>
<td>WPP nodes and edges in test applications.</td>
<td>301</td>
</tr>
<tr>
<td>TABLE 12.2</td>
<td>WPP independence experiment results.</td>
<td>303</td>
</tr>
<tr>
<td>TABLE 12.3</td>
<td>WPP false positive experiment results.</td>
<td>304</td>
</tr>
<tr>
<td>TABLE 12.4</td>
<td>WPP resistance experiment results.</td>
<td>305</td>
</tr>
<tr>
<td>TABLE 12.5</td>
<td>TaNaMM, K-Gram, and WPP independence results.</td>
<td>306</td>
</tr>
<tr>
<td>TABLE 12.6</td>
<td>TaNaMM, K-Gram, and WPP false positive results.</td>
<td>307</td>
</tr>
<tr>
<td>TABLE 12.7</td>
<td>TaNaMM, K-Gram, WPP resistance results.</td>
<td>308</td>
</tr>
<tr>
<td>TABLE E.1</td>
<td>TaNaMM independence experiment similarity results.</td>
<td>332</td>
</tr>
<tr>
<td>TABLE E.2</td>
<td>TaNaMM false positive experiment similarity results.</td>
<td>332</td>
</tr>
<tr>
<td>TABLE E.3</td>
<td>TaNaMM resistance experiment results for IterFact.</td>
<td>333</td>
</tr>
<tr>
<td>TABLE E.4</td>
<td>TaNaMM resistance experiment results for IterFib.</td>
<td>333</td>
</tr>
<tr>
<td>TABLE E.5</td>
<td>TaNaMM resistance experiment results for RecurFact.</td>
<td>333</td>
</tr>
<tr>
<td>TABLE E.6</td>
<td>TaNaMM resistance experiment results for RecurFib.</td>
<td>334</td>
</tr>
<tr>
<td>TABLE E.7</td>
<td>TaNaMM resistance experiment results for decode.</td>
<td>334</td>
</tr>
<tr>
<td>TABLE E.8</td>
<td>TaNaMM resistance experiment results for FFT.</td>
<td>334</td>
</tr>
<tr>
<td>TABLE E.9</td>
<td>TaNaMM resistance experiment results for wc.</td>
<td>335</td>
</tr>
<tr>
<td>TABLE E.10</td>
<td>K-Gram independence experiment similarity results.</td>
<td>335</td>
</tr>
<tr>
<td>TABLE E.11</td>
<td>K-Gram false positive experiment similarity results.</td>
<td>336</td>
</tr>
<tr>
<td>TABLE E.12</td>
<td>K-Gram resistance experiment results for IterFact.</td>
<td>336</td>
</tr>
<tr>
<td>TABLE E.13</td>
<td>K-Gram resistance experiment results for IterFib.</td>
<td>336</td>
</tr>
<tr>
<td>TABLE E.14</td>
<td>K-Gram resistance experiment results for RecurFact.</td>
<td>337</td>
</tr>
<tr>
<td>TABLE E.15</td>
<td>K-Gram resistance experiment results for RecurFib.</td>
<td>337</td>
</tr>
<tr>
<td>TABLE E.16</td>
<td>K-Gram resistance experiment results for decode.</td>
<td>337</td>
</tr>
<tr>
<td>TABLE E.17</td>
<td>K-Gram resistance experiment results for fft.</td>
<td>338</td>
</tr>
<tr>
<td>TABLE E.18</td>
<td>K-Gram resistance experiment results for wc.</td>
<td>338</td>
</tr>
<tr>
<td>TABLE E.19</td>
<td>WPP independence experiment results.</td>
<td>338</td>
</tr>
<tr>
<td>TABLE E.20</td>
<td>WPP false positive experiment results.</td>
<td>339</td>
</tr>
<tr>
<td>TABLE E.21</td>
<td>WPP resistance experiment results.</td>
<td>339</td>
</tr>
</tbody>
</table>
ABSTRACT

Prior to the general availability of high speed Internet, the spread of pirated software required the transfer of a physical copy like a disk, which limited the rate at which illegal software could be distributed. The low transfer rate restricted software piracy to levels which producers found acceptable because the associated losses could be absorbed. Large scale cases of piracy were rare and when they did occur the legal system provided suitable retribution. However, recent advances in computer technology have made the need for a physical copy obsolete. Piracy is now a widespread, decentralized problem in which millions of individuals take part. Without technical means of identifying pirated software, the protection afforded by the legal system is no longer easy to enforce or cost effective.

The research in this dissertation addresses the threat of software piracy through the exploration of two techniques: software watermarking and software birthmarking. Neither of these techniques can be used to prevent software theft entirely. Instead, they are used to detect occurrences of theft after the fact. One of the limiting factors of the protection provided by the legal system is that it cannot be used to identify an incidence of piracy. Software watermarking and birthmarking fill this gap, thus providing complimentary protection to the established legal protection. In this research, we analyze the state of the art in both software watermarking and birthmarking and we propose a novel scheme in each of the areas which make significant improvements over existing techniques.
CHAPTER 1

INTRODUCTION

The problem of protecting software from illegal copying and redistribution has been the focus of considerable research motivated by billions of dollars in lost revenue each year [15]. Recent increases in transfer speeds, ease of access to the Internet, and the widespread usage of peer-to-peer networks has eliminated the need for physical copies of software. To compound the problem, software is being legally distributed in platform independent formats, such as Java bytecode and Microsoft’s Intermediate Language (MSIL). These formats closely resemble source code and contain information which aids in manipulation and reverse engineering. Such formats make it easier for software pirates to bypass license checks. Additionally, the ease of reverse engineering benefits the unscrupulous programmer who wants to decrease production time and get an edge on the competition by stealing crucial algorithmic secrets.

There are three major threats recognized against the intellectual property contained in software. *Software tampering* is the illegal modification of a program to circumvent license checks, to obtain access to digital media protected by the software, etc. *Malicious reverse engineering* is the extracting of a piece of a program to reuse it in one’s own program. *Software piracy* is the unauthorized use, duplication or sale of legally available software. Currently no single mechanism exists to prevent these three major threats. A variety of mechanisms have been explored through legal, ethical and technical means. The legal approach relies on a variety of laws such as copyrights, patents, trademarks, and trade secrets. In the United States, there are ramifications associated with violating these laws, such as statutory damages of up to $150,000 for each program copied [83]. However, it is often difficult to detect that a violation has occurred. These laws offer limited effectiveness by relying on the fear of consequences, however they offer little deterrence for those who feel that
detection will be difficult.

Because the legal protections currently have limited effectiveness a variety of technical approaches are being explored to prevent and detect individual cases of piracy. Each of the techniques are designed to address one of the three major threats. By combining the techniques, a stronger defense which provides multiple levels of protection can be achieved. Such techniques include, but are not limited to, code obfuscation [36] which aids in the prevention of reverse engineering by making the program difficult to read and understand; software tamperproofing [26] which enables the software to detect alterations and take appropriate action; software watermarking [18, 30, 35, 42, 69, 86, 93, 100] which is used to deter piracy by embedding a unique identifier in the software; and software birthmarking [74, 76, 95] which is used to detect program theft through unique program characteristics.

1.1 Contributions

Instead of trying to prevent software theft, the research in this dissertation focuses on two approaches which make it possible to detect individual occurrences of theft. In this research we analyze the state of the art in both software watermarking and birthmarking. This analysis includes an extensive discussion, evaluation, and comparison of the most comprehensive collection of published software watermarking algorithms to date. To the best of our knowledge, no other single publication provides such an analysis. Additionally, we propose a novel technique in each of the areas which make significant improvements over existing techniques. Both of these approaches address the most limiting aspect of the current protection provided by the legal system.

In the area of software watermarking we propose a technique we call Branch-Based software watermarking. The technique provides the capabilities to embed an authorship mark, a fingerprint mark, and tamper detection. The Branch-Based watermarking algorithm was originally developed to protect native executables and relied upon features specific to the x86 processor. However, we have also developed
an implementation which maintains the essence of the algorithm using features specific to Java bytecode and the Java Virtual Machine. The Branch-Based software watermarking algorithm incorporates ideas from code obfuscation and software tamperproofing which allows us to make several improvements over previously proposed techniques:

1. We are able to simultaneously provide proof of authorship and trace the source of an illegal distribution.

2. The algorithm demonstrates a significantly higher level of resilience to attack without significant overhead.

3. Features of the algorithm make it possible to distribute pre-packaged, fingerprinted software which is linked to the consumer.

In the area of software birthmarking we extend the definition of software birthmarking by proposing the first known dynamic birthmarking technique. The technique, which is language independent, constructs a birthmark by capturing a representation of the program’s dynamic control flow as it executes on a particular input. The technique is based on the idea that the dynamic characteristics of a program are harder to modify than static characteristics. The Whole Program Path birthmark makes several contributions over existing software birthmarking techniques:

1. It is the first known dynamic birthmarking technique.

2. The technique demonstrates a significantly higher level of resistance to semantics-preserving transformations than the previously proposed static birthmarking techniques.

3. Because of the high level of resistance, the birthmark can be used to identify program theft even when an embedded watermark has been destroyed by a program transformation.
1.2 Organization

The work in this thesis is divided into four main parts. Part I consists of background material. We begin with a general discussion of software piracy in Chapter 2 which includes a treatment of both deterrent and preventive solutions. Chapter 3 defines terminology used in the descriptions of the protection techniques. The implementations of the protection techniques described in Parts III and IV require a general knowledge of Java bytecode, the Java Virtual Machine, and x86 assembly language programming. Necessary background information regarding these topics is presented in Appendix A.

Part II focuses on the SandMark tool. SandMark is a tool designed to study various software protection techniques including software watermarking and birthmarking. Each of the techniques described in Parts III and IV have been implemented within the framework and evaluated using the tools designed specifically for studying the strength of software protection techniques.

Part III is dedicated to the exploration of software watermarking as a technique for discouraging and detecting software theft. Chapter 5 provides a general overview of software watermarking and how it is used. This includes a formal definition, an overview of the characteristics used to classify a software watermarking algorithm and an outline of the evaluation framework used to evaluate each of the software watermarking algorithms discussed. In Chapters 6 and 7 we describe existing software watermarking techniques and provide an analysis of their strengths and weaknesses. The SandMark tool has provided us with insight on the various software watermarking algorithms. From this we have made extensions to a few of the algorithms in an attempt to improve on their weaknesses. In Chapter 8 we present these extensions. Our novel watermarking algorithm, Branch-Based software watermarking, is described and evaluated in Chapter 9.

Finally, Part IV addresses the use of software birthmarking for theft detection. Chapter 10 presents a general overview of software birthmarking including a formal definition and the properties used in strength evaluation. We also detail the frame-
work used to evaluate the techniques described in Chapters 11 and 12. In Chapter 11, we describe the two previously published static birthmarking techniques and evaluate their strengths. Our novel dynamic birthmarking algorithm, Whole Program Path software birthmarking, is described and evaluated in Chapter 12.

The research in this thesis is the culmination of work produced over several years. Certain pieces of this work have appeared in previously published forms. Those publications which relate to the work in this dissertation are listed below.

- Christian Collberg, Ginger Myles, Andrew Huntwork. SandMark – A Tool for Software Protection Research. IEEE Security and Privacy, Vol. 1, Num. 4, July/August, pgs. 40-49, 2003. In this article we present an overview of the SANDMARK tool and its capabilities. The text in Chapter 4 draws on this article, but is updated to reflect the current architecture.

- Ginger Myles and Christian Collberg. Software Watermarking Through Register Allocation: Implementation, Analysis, and Attacks. In Information Security and Cryptology - ICISC 2003: 6th International Conference, Vol. 2971, Lecture Notes in Computer Science, pgs. 274-293, Seoul, Korea, 2003. Springer-Verlag. In this article we explore the application of the graph coloring-based watermarking algorithm proposed by Qu and Potkonjak to software watermarking. Through the use of SANDMARK we perform the first empirical evaluation of the algorithm. We present the strengths and weaknesses of the proposed algorithm when it is directly applied to software watermarking. In addition, we present the modifications necessary to make it a viable algorithm. Work from this article is discussed in Chapters 7 and 8.

- Ginger Myles. Using Software Watermarking to Discourage Piracy. ACM Crossroads, Vol. 10, Num. 3, Spring 2004. This article provides an overview of software watermarking and how it can be used in the battle against software piracy. The article is directed at an audience who is unfamiliar with software-based approaches to software protection. Material from this article is used in Chapters 2, 5, and 6.
• Ginger Myles and Christian Collberg. Software Watermarking via Opaque Predicates: Implementation, Analysis, and Attacks. 7th International Conference on Electronic Commerce Research, Dallas, TX, June 10-13, 2004. In this article we analyze an algorithm originally proposed by Geneviève Arboit in A Method for Watermarking Java Programs via Opaque Predicates. This was the first known empirical evaluation of the algorithm. In addition, we present a novel extension to the algorithm which uses a dynamic recognition technique. Both static and dynamic versions of the algorithm are implemented and evaluated within the SandMark framework. Portions of Chapters 7 and 8 draw directly from the work presented in this article.

• Ginger Myles and Christian Collberg. Detecting Software Theft via Whole Program Path Birthmarks. In Information Security, 7th International Conference, ISC 2004, Vol. 3225, Lecture Notes in Computer Science, pgs. 404-415, Palo Alto, CA, USA, 2004. Springer-Verlag. In this article we propose the concept of a dynamic birthmark. Based on this concept we present and empirically evaluate the novel software birthmarking technique Whole Program Path birthmarking. In addition, we evaluate the static birthmarking technique proposed by Tamada et al. and show that it is easily defeated by current code obfuscation tools. The work in Part IV is based on material presented in this article.

• Ginger Myles and Christian Collberg. k-gram Based Software Birthmarks. In Proceedings of the 2005 ACM symposium on Applied Computing, pgs. 314-318, Santa Fe, New Mexico, USA, 2005. In this article we present and empirically evaluate a novel birthmarking technique which uniquely identifies a program through instruction sequences. In addition, we provide an evaluation of the static birthmarking technique proposed by Tamada et al. so as to make a comparison between our technique and theirs. The material in Chapters 10 and 11 draws on the work in this article.

• Ginger Myles. Preventing Piracy within the Video Game Industry. Interna-
tional Digital Media and Arts Journal, Vol. 2, Num. 1, Spring 2005. In this article we discuss current techniques for preventing piracy within the video game industry. It is an overview article which presents both hardware and software-based approaches to piracy prevention. The article is aimed at an audience who is unfamiliar with software protection techniques. Some material from this article is used in Chapter 2.

- Ginger Myles, Christian Collberg, Zachary Heidepriem, Armand Navabi. The Evaluation of Two Software Watermarking Algorithms. Software: Practice and Experience, Vol. 35, Num. 10, pgs. 923-938, 2005. In this article we analyze the effectiveness of two different software watermarking algorithms. The first is an algorithm proposed by Monden et al. and the second an algorithm proposed by Davidson and Myhrvold. To the best of our knowledge, this is the first implementation and empirical evaluation of these algorithms with respect to a set of properties such as bit-rate, stealth, and resilience to attack. A portion of Chapter 7 draws from the material presented in this article.

- Ginger Myles and Hongxia Jin. Self-Validating Branch-Based Software Watermarking. In 7th International Workshop on Information Hiding, Vol. 3727, Lecture Notes in Computer Science, pgs. 342-356, Barcelona, Spain, 2005. Springer-Verlag. In this article we present and evaluate a novel dynamic software watermarking algorithm, Branch-based software watermarking. The material in Chapter 9 draws directly from this article.

- Hongxia Jin, Ginger Myles, and Jeff Lotspeich. Towards Better Software Tamper Resistance. Information Security: 8th International Conference, Lecture Notes in Computer Science, Vol. 3650, pgs. 417-430, 2005. Springer-Verlag. In this article we propose a tamper resistance technique which provides both on- and offline tamper detection. In the offline approach the software dynamically detects tampering and causes the program to fail using a technique similar to that used in the Branch-based software watermarking technique. The material in this article is not directly represented in this dissertation. However,
the offline tamper detection technique is related to the software watermarking technique presented in Chapter 9.

- Ginger Myles and Stefan Nusser. Content Protection for Games. IBM Systems Journal, Vol. 45, Num. 1, pgs. 119 - 143, 2006. In this article we review the state of the art in copy protection technologies for games. We show the capabilities of currently deployed solutions and focus on shortcomings and problems. In an attempt to address some of the open issues in content protection for games we present two novel protection approaches. The first approach uses Branch-based software watermarking. In the second approach, we propose the use of current physical media copy protection technologies for gaming content. Finally, we describe how the proposed protection mechanisms can be deployed for video game copy protection through five scenarios. Some material in Chapter 9 overlaps with the material in this article.
I

BACKGROUND
CHAPTER 2

SOFTWARE PIRACY

Software piracy can be defined as using, copying, selling, and/or distributing computer software without prior permission of the producer(s). Throughout the world such activity is known to be illegal yet it is still undertaken. It is estimated that $29 billion of the total $80 billion of software installed on computers was actually installed illegally [15]. Piracy rates in 2003 ranged from 92% in Vietnam and China to 22% in the United States with a global piracy rate of 36%.

There are a variety of factors which contribute to what is, by some, considered to be a high rate of piracy. Such factors include the degree to which protection technologies are used as well as the cost of any penalties incurred if caught. These penalties could include damage of reputation, fines, and the cost of violating an ethical code. Each individual weighs these factors to determine the cost of pirating. An individual is likely to pirate if the potential consequences are less than either the value of the program to the individual or the purchase price of the software. Currently many consumers are aware of the difficulty in enforcing anti-piracy laws and thus, to the user, the cost of pirating is quite low.

Piracy rates differ between different parts of the world, with the lower rates occurring in western countries. It is speculated that the higher rate in non-western cultures can be partially attributed to a shift in emphasis from individual achievement to community responsibility [57]. The non-western belief is that the entire community should enjoy and benefit from a good idea, not just a few individuals. A second factor which could be contributing to high piracy rates in some countries is simply that many people cannot afford to legally obtain the software.

The Business Software Alliance (BSA) defines 5 common types of software piracy [14]:

1. Freeloading
2. Reverse engineering
3. Bootlegging
4. Counterfeiting
5. Software piracy
1. **End-user piracy** occurs when a user reproduces copies of software without authorization. It can manifest itself in one of the following forms:

   - A user obtains a single licensed copy and uses it to install the software on multiple computers.
   - The disks used to install the software are duplicated and then distributed.
   - A user purchases and installs an upgrade without previously having a legal version.
   - Within a commercial environment, employees use software with an academic license.

2. **Client-Server overuse** occurs when a program is installed on a network and is simultaneously used by more people than the license entitles.

3. **Internet piracy** occurs when illegal copies of software are made available on the Internet either free of charge or for a fee. Examples of such sites include:

   - Sites which make software available for free or by exchanging uploaded programs.
   - Auction sites that offer illegal software.
   - Peer-to-peer networks which enable the transfer of illegal software.

4. **Hard-disk loading** occurs when illegal software is installed on a new computer and sold. This activity often occurs when a business is trying to cut costs to make their products more attractive.

5. **Software counterfeiting** occurs when copyrighted material is illegally duplicated and sold with the intent that the material pass as the original.

   In the United States, and many other countries, there are legal ramifications associated with software piracy. A violation of the United States Copyright Act can result in statutory damages of up to $150,000 for each program copied. Enforcement of the laws used to protect software is a costly process in which it is often difficult to
Figure 2.1: A rapid increase in illegal software sales on Internet auction sites was recorded between 1999 and 2000 [90].

identify the person actually guilty of the violation. All too often the unsuspecting end user is the one who suffers either through being charged with the violation or by being tricked into buying fraudulent material.

Software counterfeiters have become extremely good at passing their software off as legitimate and selling it on Internet auction sites. When a person unknowingly purchases and uses the illegal software it can be difficult to trace the software back to the guilty pirate. Because of the high percentage of illegal software being sold on Internet auction sites, many companies, such as eBay, have placed restrictions on the sale of software. Figure 2.1 shows the drastic increase of illegal sales between 1999 and 2000.

To tackle software piracy a variety of solutions have been proposed. These solutions can be classified as either deterrent or preventive. Deterrent solutions rely on the fear of the consequences of getting caught. The solution is successful if an individual abstains criminal behavior due to the perceived threat or fear of sanctions. Preventive solutions make use of current technology to increase the cost of the actual act of piracy. These solutions can either be hardware- or software-based and include such techniques as tamperproof CPUs and software encryption.
Deterrent and preventive solutions will be explored further in the following sections.

The issues associated with software piracy are not obvious to everyone, which could be due in part to the non-exclusionary nature of a computer application [96]. To illustrate, suppose Alice has a copy of a popular video game on her computer. Alice can make a copy of the game and give it to Bob so he can play it on his computer. Now both Alice and Bob own copies of the game which makes the computer game non-exclusionary. On the other hand, Alice and Bob’s computers are exclusionary objects because only one of them can own each computer at a time. The exclusionary nature of the physical computer makes it clear who the property belongs to. This is not the case with intellectual property such as software.

Even though the unethical nature of software piracy might not be obvious, the concerns are certainly not new. One of the major concerns with published literature, such as articles or books, is plagiarism. Within the software industry plagiarism is also a concern, but identifying and proving that a section of an application is stolen is far more difficult than with a published piece of literature. The difficulty in detecting software theft can mainly be attributed to the format in which software is distributed. For example, in the case of source code theft, the stolen code could be compiled using a different compiler which will yield an executable that looks different than the original. In addition, the economic impact for the company whose application was stolen can be severe. Software companies often make a significant portion of their revenue prior to the release of a competitor’s product. If a portion of their application is stolen the competitor is able to decrease production time and enter the market sooner. The second ethical issue is the illegal redistribution of the software. It is generally the case that pirated copies of software are distributed at a significantly discounted price, while still including all of the original functionality. Again, there can be an economic impact associated with this act.

The ramifications associated with piracy propagate throughout the software industry. The obvious victims are the software companies themselves. However, the more peripheral victims are not often recognized. Many pirates are undeterred by reports of financial losses suffered by software producers due to piracy. This could
be because they do not see the trickle effect of the monetary losses. Many think of software producers as large companies which generate significant revenue, forgetting that the individuals who work for those companies feel the effects of piracy through decreased job opportunities or even lost jobs. In 2002, up to 105,000 jobs, $5.3 billion in wages, and $1.4 billion in tax revenues\(^1\) were lost because of piracy in the United States alone [16].

### 2.1 Deterrent Solutions

A deterrent solution relies on an individual’s fear of getting caught and does not directly increase the cost of the actual act of pirating. It is a mechanism put in place to discourage the act of piracy by imposing sanctions if the act is carried out and detected. In the United States, deterrent solutions take form in several intellectual property rights laws. The question of how these laws can be used to protect software has been debated for many years. The difficulty in devising the proper protection is rooted in categorizing software which can be a product, a service, or even a mixture of both. Currently, four intellectual property rights laws can be applied in the protection of software. These laws include copyright, patent, trademark, and trade secret.

#### 2.1.1 United States Copyright Law

Under the 1978 United States Copyright Act, as described by Tavani [96], a work must meet three requirements to receive protection. The requirements are *originality*, *fixation*, and *expression*. In general, any work which is original, has a tangible form and is fixed in a medium can be protected under copyright. In other words, one cannot copyright an idea, but the tangible expression of the idea can be copyrighted. Unfortunately, computer software is not fixed in a tangible medium like that of literary works. This was the major difficulty which, prior to 1980, made software ineligible for copyright protection.

\(^1\)These and other software piracy figures are often disputed on the grounds that not all people who acquired pirated software would purchase legal copies.
In 1980 the law was amended to address the needs of computer software. The concept of literary work was extended to include programs, computers, and databases which exhibit authorship. The amendment defines a computer program as a “set of statements or instructions to be used directly in a computer in order to bring about certain results.” This addition has made it possible to copyright a program if it can be shown that the program contains an original expression of ideas and not simply the ideas. Additionally, the amendment ensures the protection of the source, object, and executable code.

There are two doctrines associated with the Copyright Act. The first is *fair use*. Fair use permits the limited use of another person’s copyrighted work for the purpose of criticism, comment, news, reporting, teaching, scholarship, and research. Use of the work outside of this violates the holder’s rights. The second doctrine is *first sale*. The first sale doctrine allows the purchaser of a legally obtained piece of work to sell, rent, or give away the work without obtaining permission from the copyright holder. The first sale doctrine applied to software is not as straight-forward as other works protected under the U.S. Copyright Act. The main aspect which causes confusion, and has led to contradictory decisions in the courts, is that software under the End User License Agreement (EULA) is licensed, not sold. Many EULAs specifically state that resale is prohibited. Thus, the doctrine of first sale does not apply to most software purchases.

**Digital Millennium Copyright Act**

The Digital Millennium Copyright Act (DMCA) was signed into law October 28, 1998. The DMCA made significant changes to the United States copyright law to address the changing needs associated with the digital age. One of the main focuses of the act was to address the treaties signed in December 1996 at the World Intellectual Property Organization (WIPO) Geneva Conference. The WIPO treaties required legal means against the act of circumventing anti-piracy measures and any technology which enabled the circumvention. Section 1201, circumvention of copyright protection systems, of the DMCA [4] outlines a few permitted exceptions to
the anti-circumvention provision. These include nonprofit libraries, archives, and academic institutions under special circumstances. Additionally, it is legal to reverse engineer protected software to conduct encryption research, assess product interoperability, and to test computer security systems as long as it does not constitute copyright infringement and the person legally has the right to use a copy of the software.

The anti-circumvention provisions of the DMCA have lead to a variety of unforeseen consequences which often hinder legitimate activities. For example, the DMCA has been used to stop the presentation and publication of research on security vulnerabilities in many products. Additionally, through the introduction of copy-protected CDs, the DMCA can be used to prevent the fair use doctrine [44].

2.1.2 United States Patent Law

The United States Patent Law provides legal protection to individuals who create an invention or process [57]. A patent provides the inventors with exclusive rights to make, use, or sell the invention for 17 years. There are two basic requirements for an invention or discovery to be patentable:

1. It must be new and useful or a new and useful improvement.

2. It must satisfy the following:

   • The invention must have some usefulness or utility.

   • The invention must be novel.

   • The invention must be non-obvious to a “person of ordinary skill in the art” who is familiar with the prior art.

Due to the algorithmic nature of a software program, software was initially ineligible for patent protection. The first software patent was granted in 1981 for a program which assisted in converting rubber into tires (Diamond v. Deihl). Currently about 20,000 new software patents are issued each year despite considerable debate over the appropriateness of patenting software [96].
2.1.3 United States Trademark Law

Kizza [57] describes a trademark to be a word, name, phrase, or symbol that identifies a product or service. They are often used by consumers to choose between competing products. Thus the United States Trademark Law helps ensure that the quality associated with a mark used by a business actually represents the quality expected by the consumer. The laws give the owner of a trademark the ability to prevent others from using the same or similar mark to promote their products. Under United States law there are three categories of trademarks which are protected in 10 year increments:

- Service mark: Used in the sale or advertising of a service.
- Certification mark: Used as a verifier or authentication of a product, service, or group who offer a service.
- Collective mark: Used by a group of people to indicate membership.

The only protection software actually receives through trademarks is if a pirate is deterred by the difficulty of passing off copies of well-known software.

2.1.4 United States Trade Secret Law

Unlike the three previous legal protections, United States Trade Secrets have no federal protection. All laws designed to protect trade secrets are at the state level and thus offer varying degrees of protection depending on the state. A trade secret consists of information used by a business or company which is of strategic importance in providing an actual or potential economic advantage over competitors. This information may be a formula, a design process, a device, or trade figures. Owners have exclusive rights to the secret but only for as long as the secret is maintained [57].

Applying trade secret law to software is difficult. Often what makes the software valuable is a particular technique or algorithm used. Because this information is released with the software, reverse engineering techniques can often be used to discover the secrets.
2.2 Preventive Solutions

Once the correct law is chosen there is the additional difficulty of enforcing the law. It is the responsibility of the intellectual property owner to ensure that their rights are not violated. To this end a variety of preventive solutions have been devised. These include audits of companies as well as hardware- and software-based techniques.

2.2.1 Audits

Organizations such as the Business Software Alliance (BSA) perform audits to verify that corporations are not using illegal software. An audit involves taking an inventory of all material related to the software on the computer systems. Such material includes:

- All media for installation;
- All manuals and reference documentation;
- All license documentation;
- All documents proving the legitimacy of the software, e.g. invoices.

Unfortunately, auditing does not necessarily identify an unknown software pirate or unethical programmer. It can help companies identify illegal practices by their employees or identify companies who are not purchasing the required number of licenses. However, the technique is ineffective at detecting the theft of algorithmic secrets. Additionally, the technique can identify if a company is guilty of piracy, but from the information the source of the piracy may not be obvious.

2.2.2 Hardware-Based Techniques

Special purpose hardware is commonly used in proof of ownership, to provide secure data storage, and to provide a secure execution context for security-sensitive
applications. Such hardware is typically more cumbersome for the user and more expensive for the software vendor than software-based techniques.

Dongles

A *dongle* is a hardware device distributed with software. Possession of the device proves ownership of the software. A dongle typically connects to an I/O port and computes the output of a secret function. While running, the software periodically queries the dongle. If the communication fails or the result of the query is incorrect, the software reacts appropriately. There are two major drawbacks associated with the use of a dongle: (1) cost (a single dongle costs at least $10) and (2) distribution of a dongle with software over the Internet is impractical. The dongle was once the protection technology of choice. However, the use has fallen out of favor. From a technical perspective, the dongle suffers from a major weakness in that the attack point is clearly defined. The interface to the device is a hardware interface which means that the signals passing over the interface must conform to the hardware standards. This gives the attacker an analysis advantage.

Tamperproof CPUs

*Tamperproof CPUs* aid in piracy prevention by providing a secure context and/or secure data storage. By executing the software in a secure environment the pirate is unable to gain access to the software. This technique prevents the attacker from observing the behavior of the software which means he is unable to identify portions of the software to remove. The obvious drawback to this technique is the additional cost of requiring all users to have tamperproof hardware.

Lie et al. [62] propose one such technique in which standard hardware is modified so that encrypted code can be securely executed. To accomplish this, each “XOM” chip contains a different decryption key. To execute the encrypted code the processor enters XOM mode. The instructions are then decrypted and verified in the XOM Instruction Decryption Unit. The special Decryption Unit is only used
in XOM mode so as to minimize the overhead incurred due to the hardware modifications. Prior to the processor switching from secure to normal operation mode the architectural state is secured. In this process the registers and cache are cleared and all pending writes are completed. This prevents a user from waiting until the XOM mode has completed to obtain information which could reveal the encrypted instructions. Such architectural support makes it possible to maintain algorithmic secrets. Additionally, because each XOM chip uses a different decryption key it is possible to prevent execution of unauthorized copies of the software.

**Smartcards**

*Smartcards* are used in many contexts to securely store data. For example, smartcards store cryptographic keys for use in authentication systems and channel authorizations for use in broadcast television systems. A traditional smartcard consists of an 8-bit microprocessor with ROM, EEPROM and RAM on a single chip with serial input and output. The EEPROM is used to store the secure information. Erasing this data requires a relatively high voltage, however, if the attacker can prevent the voltage from reaching the EEPROM the information will remain.

Early smartcards received their voltage from the host. Attackers were able to make use of this design to attack pay-TV systems which used smartcards to store subscription information. In the attack, all channels were initially enabled. Prior to cancelling the service, the attacker would cover the voltage contact using something as simple as tape. The voltage sent to the card never reached the EEPROM, allowing the attacker to continue to use the service without paying.

The next generation of smartcards raised the cost of attack, but still did not make it impossible. The design of these smartcards changed the source of the voltage used to reprogram the EEPROM. Attacks on this type of card can be carried out using a microscope and a laser. Anderson and Kuhn [17] describe a variety of attacks on smartcards along with the associated costs.
2.2.3 Software-Based Techniques

Software-based solutions, such as code obfuscation, software tamperproofing, software watermarking, and software birthmarking provide a number of advantages over strictly hardware-based techniques. First, the protection is cheaper to implement due to the lack of special purpose hardware. Second, many of the currently proposed hardware-based solutions have been easily attacked. Some of the attacks require specialized equipment, but many of the side-channel attacks are relatively cheap. For example, the protection provided by some smartcards can be defeated by shining a common light-bulb on the card [47]. Third, many hardware-based solutions can be difficult to deploy. It can take many years for users to upgrade their machines to ones which contain tamperproof CPUs and once the protection mechanism has been defeated it cannot be fixed without upgrading the hardware again.

Software-based techniques take a different approach than hardware-based techniques because it is generally believed that given enough time a determined adversary will be able to defeat any protection mechanism. The goal instead is to develop techniques which require “enough” time, effort, and/or resources to break such that it is less costly for the attacker to simply rewrite the software or purchase legal copies. Thus, the techniques can be used to extend the period in which no pirated copies exist, increasing the revenue for the software producers. This is especially useful for products such as video games which often have a short shelf life. Additionally, software-based techniques can be used in conjunction with specialized hardware to increase the strength or to further protect the software once the hardware protection has been defeated.

Code Obfuscation

*Code obfuscation* is a technique developed to aid in the prevention of malicious reverse engineering. An obfuscation is a semantics-preserving transformation which makes the program more difficult to understand while preserving the original functionality. The idea is to obscure the readability and understandability of the program
to such a degree that it is more costly for the attacker to reverse engineer the program than to simply recreate it or purchase a legal copy. There are three general classes of obfuscations:

**layout obfuscations** alter the information that is unnecessary to the execution of the application such as identifier names and source code formatting;

**data obfuscations** alter the data structures used by the program. For example, a two dimensional array could be folded into a one dimensional array;

**control flow obfuscations** are used to disguise the true control flow of the application, for example by inserting dead or irrelevant code, converting a reducible flow graph into a non-reducible graph, inlining methods, merging methods, and transforming loops using techniques such as loop unrolling.

The most common and simplest obfuscation is name obfuscation. The basic idea is to rename the identifiers in the program to meaningless names. For example, the method `getKey()` could be renamed to `a()`. Tyma [98] describes a technique in which method overloading is used to generate as few unique names as possible. For example, the methods `foo` and `bar` in Figure 2.2 can be renamed to `a`. However, `foobar` must have a different name than `bar` since they have the same signature. Additionally, `toString` cannot be renamed since it overrides `java.lang.Object.toString`. The same idea can be applied to fields and classes. When using renaming care must be taken with classes that are loaded by name and with fields and methods that are accessed using reflection.

Code obfuscation has many interesting applications. In addition to rendering applications more difficult to understand and reverse engineer, it can be used to protect watermarked programs from a collusive attack. In this attack, an adversary obtains several differently watermarked programs and compares them to identify the watermark. Through the use of obfuscation, differently fingerprinted programs can make the programs differ everywhere instead of only where the watermark was embedded. Obfuscation can also be used to perform a malicious attack against software watermarks, transforming the code such that the mark is unrecoverable.
Digital Rights Management

A Digital Rights Management (DRM) system is used to protect digital media from theft, unauthorized access, and illegal distribution. The media is protected through the use of encryption and a set of usage rules. The usage rules can define a variety of criteria associated with the media. For example, they can be used to define the length of time the content can be viewed or the number of times it can be opened. The rules can also define whether the content can be transferred to another user or device. Within the DRM system the content is stored in encrypted form. When needed, the content is decrypted in a secure, tamper-resistant environment. The tamper-resistant environment can be constructed using software and/or hardware-based techniques. For software-based implementations code obfuscation is often used. Additionally, code obfuscation can be used to hide a decryption key in the absence of a secure hardware device.

Software Tamperproofing

Code obfuscation is used to hide a secret while tamperproofing is used to protect the secret from alteration. For example, many programs contain license checks that prevent the user from using the software after a specific date. An attacker will attempt to locate and disable the check in order to enjoy the software for free. To prevent such attacks a software developer may tamperproof the license check such that if it is altered the program will no longer function properly.
A tamperproofing technique performs two duties. First the tamperproofing mechanism must detect that the software has been altered. Second, once detection has occurred, the mechanism must cause the program to fail. For the tamperproofing to be successful, the software failure must be stealthy and not alert the attacker to the location of the failure-inducing code. This can be accomplished by separating the detection and response mechanism in both space and time.

The first non-trivial tamperproofing algorithm was published by Aucsmith [19]. The key to the algorithm are the Integrity Verification Kernels (IVKs). These are small units of code that are responsible for performing critical program functions. Each IVK contains $2^N$ blocks of code all of equal size. Half of these blocks are located in upper memory and the other half in lower memory. With the exception of the initial block each IVK block is encrypted. The blocks are executed in a pseudo-random order determined by a key, beginning with the initial block. Once the code of the initial block has been executed the decrypt and jump function is performed. This function XORs each block in upper memory with a block in lower memory. The result of this operation is that at least one block in the lower memory has been decrypted and execution resumes at that block's code section. This process continues, alternating between upper and lower memory blocks. Within each cell an accumulation product (sum of the hash function of all executed blocks) is checked to verify that the previous cells were executed correctly and in the correct order. This step occurs just prior to the decrypt and jump function and is responsible for verifying the integrity of the program. If a problem is detected in this step appropriate action is taken which will eventually cause the program to fail.

A second tamperproofing technique was proposed by Chang and Atallah [26] and implemented for Win32 executables. In this algorithm tamper protection is achieved by inserting a network of guards into the program. The network establishes a check and balance system by assigning different tasks to the guards. For example, one guard may checksum a section of code while another repairs it or two guards may check the integrity of each other. Through this network of guards the algorithm is able to verify if a section of code has been tampered with.
These types of tamperproofing techniques work well for binary executables but are difficult to implement for type-safe distribution formats such as Java bytecode. While it is possible to encrypt a Java class file and use a special classloader to load and decrypt it, it is impossible to do this in a stealthy way. It is always possible for an adversary to intercept the decrypted bytecodes at the point where they are handed off to the Java Virtual Machine for execution.

Software Watermarking

Software watermarking is used to embed a unique identifier in the program. Piracy is confirmed by proving the program contains the watermark. Watermarking can be used in one of two ways [79]. If each legal copy of the program is watermarked with the same identifier then the watermark is used as a proof of authorship. This type of mark is useful in cases where a module has been stolen and incorporated into another company’s application. By detecting the authorship mark the original creator is able to prove their software was stolen. A watermark can also be used to trace the source of the illegal distribution. In this case each legal copy of the program contains a unique identifier, the fingerprint mark, which is linked to the original purchaser. If an illegal copy is identified the watermark will uniquely identify the guilty pirate. This technique is commonly referred to as fingerprinting. Software watermarking is furthered explored in Part III.

Software Birthmarking

A software birthmark is a unique characteristic, or set of characteristics, that a program possesses and which can be used to identify the program. The general idea is that if two programs $p$ and $q$ both have similar birthmarks then it is highly likely that one is a partial or modified copy of the other. Just like software watermarks, software birthmarks are used to detect software theft. However, the techniques differ in two important ways. First, it is often necessary to add code to the application in order to embed a watermark. In the case of a birthmark, additional code is never
needed. Instead a birthmark relies on an inherent characteristic of the application to show that one program is a copy of another. Secondly, a birthmark cannot be used to prove authorship or identify the source of an illegal redistribution. Rather, a birthmark can confirm that one program is contained in or is a partial copy of another. A strong birthmark will be able to provide evidence of software theft even when code transformations have been applied to the code by a malicious adversary. The use of software birthmarks in detecting theft is the focus of Part IV.

2.3 Summary

The problem of protecting software from illegal copying and redistribution has seen considerable attention in both recent research and updated legislation. The growing concern regarding software piracy can be attributed to a variety of factors such as rich distribution formats, which preserve much of the information from the source code, and the ease of sharing over the Internet. In previous years piracy was limited by the necessity to physically transfer a piece of software on a floppy disc or CD-ROM. With the increases in bandwidth physical transfer is no longer necessary.

Currently, there are a variety of techniques in use to try to prevent, discourage, and detect theft. The pros and cons of each must be weighed by the software vendor to decide if the protection afforded by the technology is worth the additional cost. Unfortunately, no single solution is currently strong enough to completely prevent piracy. However, through a combination of techniques, application developers can better protect their products. For many companies the goal is simply to protect the software long enough that a reasonable return on the investment can be obtained. Since current technologies are limited in their protection capabilities continued research into more robust techniques is necessary.
CHAPTER 3

TERMINOLOGY

Software-based approaches to software protection applied at the assembly level regularly use transformations similar to compiler transformations to incorporate the protection mechanism. These transformations rely on data-flow analysis as well as various types of program representation. In this chapter we will present some of the compiler-related tools used in the theft detection techniques presented in future chapters. We will also present other terminology that will make it easier to understand the particular software watermarking and birthmarking techniques.

3.1 Control Flow Graph

A control flow graph (CFG) is a graphical representation of the possible flow of execution through a function [12]. The graph is composed of basic blocks and directed edges. Each basic block is a sequence of statements in which the flow of control starts at the beginning of the block and does not end until the end of the block. Within a basic block a control transfer statement can only occur as the last statement in the block. Thus, all branch instructions as well as instructions which either explicitly or implicitly throw exceptions can only occur as the last instructions of a basic block. The set of basic blocks for a sequence of instructions is constructed using the following algorithm:

1. First determine the leaders. A leader is the first instruction of a basic block.

   (a) The first instruction in the sequence is a leader.

   (b) Any instruction which is the target of a conditional or unconditional branch instruction is a leader.

1Exceptions are not a concern for all languages; however, they are prominent in Java.
(c) Any instruction which follows a conditional or unconditional branch instruction is a leader.

(d) Any instruction which can catch a thrown exception is a leader.

(e) Any instruction that follows an instruction which can explicitly or implicitly throw an exception is a leader.

2. For each leader, the basic block consists of all instructions up to the next leader or the end of the sequence.

The CFG is constructed by adding directed edges between the basic blocks. An edge is added between two basic blocks $B_1, B_2$ if:

1. There is an unconditional or conditional branch instruction from the last instruction in $B_1$ to the first instruction in $B_2$.

2. $B_2$ immediately follows $B_1$ in the instruction sequence and $B_1$ does not end in an unconditional branch instruction.

3. The last instruction in $B_1$ can throw an exception and the first instruction in $B_2$ is the handler that can catch the exception.

Additionally, a CFG has 2 special blocks: start and exit. The start block is where execution begins for the function and the exit terminates for the function. Figure 3.1 illustrates the construction of a control flow graph for a method which iteratively computes a factorial.

### 3.2 Graph Coloring Register Allocator

Register allocation and assignment are important parts of the code-generation pass of a compiler. The idea is to select the local variables that could best benefit from residing in a register (the allocation problem) and then decide in which register they should reside (the assignment problem).

To prepare for register allocation, a compiler will compute the live range of each variable. These are the locations in a procedure between which the variable is first
Figure 3.1: Java bytecode of a method which iteratively computes a factorial and the corresponding control flow graph.

assigned to and when it is last used. Two variables with intersecting live ranges cannot be assigned to the same register. Thus, the register allocation problem is reduced to assigning variables to registers such that variables with overlapping live ranges are assigned to different registers. Typically, compilers will attempt to minimize the number of registers used by a procedure, while maximizing performance. The number of available registers differ from architecture to architecture, from eight for the x86, thirty-two for most RISC processors, to 65,535 for the Java Virtual Machine.

Many compilers will model the register allocation problem as a graph coloring problem. Optimization graph coloring is defined as follows:

**Definition 1** (Optimization Graph Coloring). Given a graph $G(V, E)$ color the graph with as few colors as possible such that no two adjacent vertices are colored with the same color.

Graph coloring is NP-complete but many simple heuristic algorithms that work well for register allocation have been developed.
\[ \begin{align*}
  v1 & := a \times a \\
  v2 & := a \times b \\
  v3 & := 2 \times v2 \\
  v4 & := v1 + v2 \\
  v5 & := b \times v3
\end{align*} \]

(a) Sample Code

(b) Interference Graph

Figure 3.2: Sample code and corresponding interference graph.

The graph coloring problem is applied to register allocation by constructing an interference graph [71] that models the variables and the live range interferences of a procedure. The graph has one vertex per local variable and an edge between two vertices when the corresponding variables’ live ranges interfere. As an example consider the sample code and the corresponding interference graph in Figure 3.2. Since the code has five variables, the graph has five nodes. The graph has an edge \( v1 \to v2 \) since variables \( v1 \) and \( v2 \) are live at the same time. In the interference graph, node colors (\( R_1, R_2, \ldots \)) indicate the register the corresponding variable has been assigned to. There are often many different possible colorings. Figure 3.2 shows one such assignment: variables \( v1, v4, \) and \( v5 \) are assigned to register \( R_1; v2 \) is assigned to \( R_2; \) and \( v3 \) is assigned to \( R_3. \) A graph coloring register allocator is used in a technique proposed by Qu and Potkonjak to embed a watermark in a program. The technique is described in Chapter 7.

### 3.3 Dominator Set

The dominator relationship is useful in finding various types of paths in a CFG, e.g., natural loops. Within a CFG, a block \( d \) dominates a block \( n, \) if every execution path from the start block to \( n, \) passes through \( d. \) For example, in Figure 3.1, block 2 dominates block 3, but block 3 does not dominate block 4 because the loop may not ever get executed. For each block \( b \) in the CFG, all blocks which dominate \( b \) form the dominator set for that block. The dominator set for block 3 in Figure 3.1
consists of blocks \textcolor{red}{\textbf{start}}, 1, 2, and 3. Dominator information is of particular interest in Chapter 9 where it is used to identify loops and the deterministic path through a function.

One technique for identifying a loop in a CFG is to first identify the \textit{backedge}. An edge is considered a backedge if the head dominates the tail. If \( a \rightarrow b \) is an edge, \( b \) is the head and \( a \) is the tail. The edge \( 3 \rightarrow 2 \) in Figure 3.1 is a backedge because block 2 dominates block 3. To identify the remaining blocks in the loop we iteratively consider the predecessors of the blocks known to be in the loop until all possible blocks have been considered.

The \textbf{Branch-based} software watermarking algorithm in Chapter 9 requires that only instructions along the deterministic path be used in the embedding. The deterministic path is the sequence of basic blocks which will be executed each time the function executes. Thus, it is necessary to identify those blocks which are on every execution path from the \textit{start block} to the \textit{exit block}. This is precisely the definition of the dominator set of the \textit{exit block} in the function’s CFG. The deterministic path for the function in Figure 3.1 consists of blocks \textcolor{red}{\textbf{start}}, 1, 2, 4, and \textcolor{red}{exit}.

3.4 Opaque Predicates

\textit{Opaque predicates} were first presented by Collberg et al. [37] as a technique to aid in code obfuscation and later incorporated in software watermarking techniques [18, 69]. Informally, opaque predicates are inserted to make it difficult for an adversary to analyze the control-flow of the application. This makes it more difficult to identify that certain portions of the application are superfluous. For example, an opaque predicate can by used to disguise the fact that a dummy method is never invoked or that a block of code is never executed.

\textbf{Definition 2 (Opaque Predicate).} A predicate \( P \) is opaque at a program point \( p \), if at point \( p \) the outcome of \( P \) is known at embedding time. If \( P \) always evaluates to \textcolor{red}{\textbf{True}} we write \( P^T_p \), for \textcolor{red}{\textbf{False}} we write \( P^F_p \), and if \( P \) sometimes evaluates to \textcolor{red}{\textbf{True}} and sometimes to \textcolor{red}{\textbf{False}} we write \( P^?_p \) [37].
\[
\forall x, y \in \mathbb{Z} \quad 7y^2 - 1 \neq x^2
\]

\[
\forall x \in \mathbb{Z} \quad 2 \mid \left\lfloor \frac{x^2}{2} \right\rfloor
\]

\[
\forall x \in \mathbb{Z} \quad 2 \mid x(x + 1)
\]

\[
\forall x \in \mathbb{Z} \quad x^2 \geq 0
\]

\[
\forall x \in \mathbb{Z} \quad 3 \mid x(x + 1)(x + 2)
\]

\[
\forall x \in \mathbb{Z} \quad 7 \not\mid x^2 + 1
\]

\[
\forall x \in \mathbb{Z} \quad 81 \not\mid x^2 + x + 7
\]

\[
\forall x \in \mathbb{Z} \quad 19 \not\mid 4x^2 + 4
\]

\[
\forall x \in \mathbb{Z} \quad 4 \mid x^2(x + 1)(x + 1)
\]

Table 3.1: Number-theoretically true opaque predicates.

The definition of opaque predicate can be extended to define an opaque method.

**Definition 3** (Opaque Method). A boolean method \( M \) is opaque at an invocation point \( p \), if at point \( p \) the return value of \( M \) is known at embedding time. If \( M \) always returns the value \textbf{True} we write \( M^T_p \), for \textbf{False} we write \( M^F_p \), and if \( M \) sometimes returns \textbf{True} and sometimes \textbf{False} we write \( M^?_p \).

The key challenge to using opaque predicates or opaque methods is to design them in such a way that they are resilient to various forms of analysis. If an adversary can easily decipher the value of an opaque predicate it provides very little protection for the software. A variety of techniques based on number-theoretic results, pointer aliasing, and concurrency have been suggested for the construction of opaque predicates [37]. In addition to the number-theoretic results, Arboit also suggests a technique for constructing a family of opaque predicates through the use of quadratic residues. Table 3.1 lists nine example number-theoretic opaque predicates. None of these opaque predicates are considered resistant to analysis, but they are presented here to illustrate the concept.

### 3.5 Branch Function

Branch functions were originally proposed as part of an obfuscation technique
used to disrupt static disassembly [64]. The original obfuscation technique converted unconditional branch instructions to a call to a branch function which is inserted in the program. The sole purpose of the branch function is to transfer the control of execution to the instruction which was the target of the unconditional branch. The branch function can be designed to handle any number of unconditional branches. We have devised an extension so that conditional branches can be handled as well. When this idea is applied to the x86 instruction set all jmp, call, and jcc instructions can be converted to a call to a single branch function. Figure 3.3 illustrates the general idea of the branch function.

A generalized branch function is responsible for choosing the correct target based on the call location. There are a variety of ways to accomplish such a task. An example, proposed by Linn and Debray [64], is to first identify the branch instructions and construct a mapping between the branch instructions and the targets, \( \theta = \{ j_1 \rightarrow t_1, j_2 \rightarrow t_2, \cdots, j_n \rightarrow t_n \} \). Second a perfect hash function is used to assign a unique identifier to each of the branch locations, \( h = \{ j_1, j_2, \cdots, j_n \} \rightarrow \{ 1, 2, \cdots, m \} \) where \( n \leq m \). Finally, a table \( T \) is constructed in the data section of the binary which lists the displacements for each \((j_i, t_i)\) pair. The displacements are stored in the table.
such that $T[h(j_i)] = t_i - j_i$. When the branch function executes, the perfect hash function is applied to the return address to compute $h(j_i)$. The slot $h(j_i)$ is accessed in $T$ to obtain the displacement to the target. The displacement is then added to the return address and the branch function transfers execution to the branch target.
II

SANDMARK
CHAPTER 4

THE SANDMARK TOOL

SANDMARK is a research tool designed to study a plethora of software-based software protection techniques. The ultimate goal in developing the tool is to implement and evaluate all known software-based techniques for software protection. The tool contains algorithms for code obfuscation, software watermarking, and software birthmarking as well as tools that aid in evaluating the strength of these algorithms. By combining the various protection techniques in a single tool it becomes easy to combine algorithms, to evaluate the effectiveness and performance overhead of the algorithms, and to launch manual and automated attacks against the algorithms. Additionally, the infrastructure is designed so that new algorithms can easily be added.

Through a description of SANDMARK’s overall design and capabilities we will show how the tool can be used to test and evaluate the software protection algorithms. The SANDMARK tool is used in Chapters 7-9 to evaluate the various software watermarking algorithms discussed. It is also used in Chapters 11 and 12 to evaluate the software birthmarking algorithms.

4.1 System Architecture

The current implementation of SANDMARK includes a variety of code obfuscation, software watermarking, and software birthmarking algorithms. The implementation has been a team effort including 28 students, one professor, and one staff programmer over several years. The tool itself is implemented in Java and operates on Java bytecode. To manipulate Java class files SANDMARK depends on the BCEL bytecode editing library from the Apache project [1].

The SANDMARK tool supports both a graphical user interface (GUI), to allow
Figure 4.1: The SANDMARK graphical user interface.

novices to easily experiment with the different protection algorithms, and a command line interface. The SANDMARK GUI, illustrated in Figure 4.1, contains 10 panels, one for each of the main operations performed by the tool. These operations include dynamic watermarking, static watermarking, obfuscation, optimization, bytecode comparison, bytecode viewing, decompilation, automated protecting, static birthmarking, and dynamic birthmarking.

The SANDMARK system architecture is designed to make it easy for researchers to implement novel software protection algorithms, evaluate the algorithms, and compare them against the existing techniques. This is accomplished through a plugin architecture in which each algorithm is implemented by extending a very simple interface. For example, a plugin that implements a dynamic watermarking algorithm extends the DynamicWatermarker class. The plugin architecture enables runtime discovery as well as full integration in the GUI, the regression and performance test suites, and other infrastructure components. The abstract class
**sandmark.Algorithm** in Figure 4.2 allows **sandMark** to properly display, configure, and determine the effects of any plugin which is derived from the class.

To illustrate the plugin architecture consider the simple obfuscation in Figure 4.3, which changes all private and protected methods to public methods. The algorithm iterates over the methods in the class, unsetting the private and protected access flags and then setting the public access flag for the method. The **apply** method in this class is called by the **sandMark** infrastructure to obfuscate the indicated class. The **getMutations** method indicates to **sandMark** that after this obfuscation has been applied, method access flags will have been changed. This information is useful for the quick protect algorithms in which subsequent algorithms should not undo the protection of previous algorithms. The quick protect algorithms are further discussed in Section 4.3. The other methods in the class provide information to the user regarding the particular obfuscation.

In addition to the plugin architecture, algorithm implementation is made easier through a simplified representation of the Java application and the component classes, methods, fields, and files. The **program objects** are built on top of the BCEL bytecode editing library. The objects provide wrappers for BCEL methods, as well as additional functionality not found in BCEL. For example, given a method object, the control flow graph or interference graph can be obtained through a single method call.

To evaluate the performance and correctness of the protection algorithms, **sandMark** contains a test suite comprised of the standard benchmark suites SpecJVM [39] and CaffeineMark [91], and the Kaffe regression tests [56]. The bytecode viewing and comparison tools can be used to evaluate and attack watermarking and obfuscation algorithms by simulating manual attacks. These two tools are described in detail in Section 4.4.2 and 4.4.3, respectively.
package sandmark;

/**
 * The Algorithm class encapsulates common characteristics of all
 * Sandmark algorithms. Most of this information is about the
 * description of the algorithm, while some of the information
 * is about the algorithm's interaction with other Sandmark algorithms.
 */
public abstract class Algorithm {
    public abstract String getShortName();
    public abstract String getLongName();
    public abstract java.lang.String getAlgHTML();
    public abstract java.lang.String getAlgURL();
    public abstract String getAuthor();
    public abstract String getAuthorEmail();
    public abstract String getDescription();

    // Gets the ConfigProperties object which specifies the parameters
    // for this run of the algorithm.
    public abstract sandmark.util.ConfigProperties getConfigProperties();

    // Specifies the types of modifications that the algorithm makes.
    public abstract sandmark.config.ModificationProperty[] getMutations();

    // Specifies a list of properties of algorithms that must be run
    // on the target code before this algorithm is run.
    public abstract sandmark.config.RequisiteProperty[] getPrerequisites();

    // Specifies a list of properties of algorithms that must be run
    // on the target code after this algorithm is run.
    public abstract sandmark.config.RequisiteProperty[] getPostrequisites();

    // Specifies a list of properties of algorithms that should be run
    // on the target code before this algorithm is run, but are not
    // necessary.
    public abstract sandmark.config.RequisiteProperty[] getPresuggestions();

    // Specifies a list of properties of algorithms that should be run
    // on the target code after this algorithm is run, but are not
    // necessary.
    public abstract sandmark.config.RequisiteProperty[] getPostsuggestions();

    // Specifies a list of properties of algorithms that cannot be run
    // on the target code before this algorithm is run.
    public abstract sandmark.config.RequisiteProperty[] getPreprohibited();

    // Specifies a list of properties of algorithms that cannot be run
    // on the target code after this algorithm is run.
    public abstract sandmark.config.RequisiteProperty[] getPostprohibited();
}

Figure 4.2: The sandmark.Algorithm class makes it possible to properly display, configure, and determine the effects of any plugin which is derived from the class.
public class SetMethodsPublic extends sandmark.obfuscate.ClassObfuscator {

    static int AND_MASK = "(org.apache.bc5.Constants.ACC_PRIVATE
            | org.apache.bc5.Constants.ACC_PROTECTED);";
    static int OR_MASK = org.apache.bc5.Constants.ACC_PUBLIC;

    public String getAuthor() {
        return "Ginger Myles";
    }

    public String getAuthorEmail() {
        return "mylesg@cs.arizona.edu";
    }

    public String getDescription() {
        return "Make all the methods in a class public";
    }

    public sandmark.config.ModificationProperty[] getMutations() {
        return new sandmark.config.ModificationProperty[] {
            sandmark.config.ModificationProperty.I_PUBLICIZE_METHODS;
        };
    }

    public String getShortName() {
        return "Publicize Methods";
    }

    public String getLongName() {
        return "Make all the methods in this class public";
    }

    public void apply([sandmark.program.Class cls | throws Exception {
        sandmark.program.Method[] methods = cls.getMethods();

        for (int i = 0; i < methods.length; i++){
            methods[i].setAccessFlags([methods[i].getAccessFlags() & AND_MASK] |
            OR_MASK);
        }
    }

    cls.mark();
    }
}

Figure 4.3: A simple obfuscation which changes all private and protected methods to public methods.
4.2 Support Code

Software protection algorithms rely on semantics-preserving transformations, making them similar to code optimization techniques in many respects. One particular similarity is in the types of analyses used to perform the transformation. To aid in implementing protection algorithms SANDMARK includes 11 static analyses as well as an extensive graph library and an opaque predicate library. The 11 static analyses include:

- **Call graph analysis**: A graph is constructed which represents the calling structure of the application. Each node in the graph represents a method. A directed edge is added between methods $m_1$ and $m_2$ if an instruction in $m_1$ invokes $m_2$.

- **Class hierarchy analysis**: A graph is constructed which represents the inheritance hierarchy of the class. Each node in the graph represents a class. There exists an edge from class $c_1$ to class $c_2$ if and only if $c_2$ extends or implements $c_1$.

- **Control flow analysis**: A method or program level control flow graph is constructed representing the possible paths of execution.

- **Def-use analysis**: Definition and use information about variables within a method. The analysis includes both def-use and use-def chains.

- **Dependency analysis**: A graph is constructed which represents the dependencies between fields and methods. Each node is either a field or a method. An edge is added from a field or method to the field or method it refers to.

- **Expression tree construction**: Expression trees are constructed representing the instructions in the basic blocks of a method.

- **Forward and backward slicing**: Depending on the type of slice a sequence of instructions is constructed which either contribute to a variable at a program point or are affected by that variable.
• **Initialized variable analysis:** Analyzes whether a particular local variable has been initialized at a particular instruction in a method.

• **Liveness analysis:** Analyzes a control flow graph to determine where a variable is live. Uses def-use analysis.

• **Stack simulation:** A utility to perform conservative evaluation of a method. The simulator enumerates all possible contexts for an instruction, tracks the local variable table and execution stack for a method.

• **Variable interference analysis:** A method level interference graph is constructed based on the live range of each variable. Uses liveness analysis in the construction.

**SAMDARK**’s support code is crucial to simple and consistent implementation of the protection algorithms. A significant portion of **SAMDARK** is dedicated to this code.

### 4.3 Protection Modules

The software protection algorithms within **SAMDARK** are split into three different modules: obfuscation, watermarking, and birthmarking. The *obfuscation module* includes 34 individual obfuscations. These obfuscations are applied at the application, class, or method level depending on the type of transformation. Class and method level obfuscations can be customized through the GUI. The default for these obfuscations is to obfuscate all of the classes or methods in the application. Figure 4.4 illustrates that it is also possible to choose pieces of the application for customized obfuscation. In this example, two of the methods from a factorial program are selected for obfuscation. Brief descriptions of the obfuscation algorithms included in **SAMDARK** can be found in Appendix B.

In addition to the individual obfuscations, **SAMDARK** includes several experimental *obfuscation executives*. An obfuscation executive is an algorithm which chooses an optimal set of transformations and program parts to obfuscate. The
Figure 4.4: Class and method level obfuscations can be customized so only selected portions of the application are obfuscated.

optimal selection is guided by the user indicating the level of overhead that can be tolerated, how much obfuscation should be added, and flagging security and performance critical parts. Figure 4.5 demonstrates the configuration options associated with an obfuscation executive. The executive obfuscates the application by repeatedly choosing a portion of the application to obfuscate, choosing an appropriate transformation, and then applying the transformation. The obfuscation executives can only be accessed through the GUI and are located on the Quick Protect panel.

The *watermarking module* within SANDMARK includes both static and dynamic algorithms of varying complexity. Currently, the tool includes three dynamic and thirteen static algorithms. Dynamic watermarking algorithms use information gathered at runtime to both embed and recognize the watermark. To gather this information SANDMARK makes use of Java’s JDI (Java Debugging Interface). JDI allows SANDMARK to perform all of the operations normally available during interactive
debugging, e.g., setting breakpoints, examining variables, and stepping through the application. The static and dynamic watermarking algorithms included in SANDMARK are explained in detail in Chapter 7.

As with the obfuscation module, the watermarking module includes one automated watermarking loop. The *watermarking loop* constructs a sequence of static watermarking algorithms such that no watermark will be destroyed during future watermark embeddings. For the watermarking loop to be able to function properly each static watermarking algorithm must accurately implement the methods `getMutations`, `getPrerequisites`, `getPostquisites`, `getPresuggestions`, `getPostSuggestions`, `getPreprohibited`, and `getPostprohibited` specified in the class `sandmark.Algorithms` (Figure 4.2). The watermarking loop is accessed through the Quick Protect panel of the GUI.

The *birthmarking module* is similar in design to the watermarking module in that the algorithms are classified as either static or dynamic. There are currently five
static algorithms and one dynamic. Dynamic algorithms compute a program’s birth-
mark based on runtime information. As with the watermarking algorithms, there is
heavy reliance on Java’s JDI to gather this information. The static algorithms can
be applied at the class or application level to detect whole program or individual
module theft. The current dynamic algorithm is only directed at detecting whole
program theft. The birthmarking algorithms included in SAndMARK are explained
in detail in Chapters 11 and 12.

4.4 Evaluation Tools

In addition to the test suite used to evaluate the correctness and overhead of the
software protection algorithms, SAndMARK supports a variety of manual attack
tools. These tools are used to simulate attacks in which the bytecode is inspected
to identify unusual code segments.

4.4.1 Software Statistics

The SAndMARK statistics module is designed to compute various complexity mea-
sures. These statistics are useful for manually evaluating the protection algorithms.
For instance, they can be used to evaluate the complexity of a program before and
after obfuscation. This will give an indication as to the overall effectiveness of the
obfuscation. The statistics can also be used to simulate an attack in which an
adversary tries to isolate a watermark through statistical anomalies.

A set of basic statistics have been implemented at the application, class and
method levels. These include:

• method/class size statistics: These statistics quantify such properties as
  method/class size in bytes, number of methods in a class, number of instance
  variables in a class, etc.

• method/class internal statistics: The method level statistics consider the num-
  ber of decision points, API calls, nested expressions, and number of param-
eters. The class level computes properties such as class cohesion and global usage of variables.

- **class external statistics**: These statistics indicate how the class relates to other classes through such properties as class coupling and number of reuses of a class.

- **method/class inheritance statistics**: The inheritance statistics consider properties such as class hierarchy nesting level, number of abstract classes, level of multiple inheritance, number of overridden methods, and number of methods inherited by a subclass.

In addition to the basic statistics, various published software complexity metrics are also implemented. These software metrics encompass different code properties of Java applications such as class inheritance hierarchy, nesting level complexity, data structure complexity, coupling between different classes/methods, information flow between classes/methods, etc. The six complexity metrics currently implemented are:

- McCabe’s [66] cyclomatic measure which calculates the number of conditional statements.

- Halstead’s [50] measure which computes the number of operators and operands.

- Munson and Kohshgoftaar’s [72] data structure metric which calculates the complexity measure based on the number of scalar and vector variables.

- Chidamber and Kemerer’s [28] object oriented complexity metric which calculates properties such as class hierarchy level, coupling between different classes, etc.

- Henry and Kafura’s [54] complexity measure which computes the information flow across different classes and methods.
• Harrison and Magel’s [51] nesting level complexity measure which calculates the depth of loop nesting.

The complexity metrics are also used by the obfuscation executive. After each round of obfuscation, the change in complexity is used to determine which parts of the program need further obfuscation.

4.4.2 View Pane

A common form of attack on protected software is to manually inspect the code to identify unusual code segments. To simulate such an attack, SANDMARK includes a view pane which can be used to view various information about an application. The application is displayed in a tree structure that illustrates the relationship between packages, classes, and methods.

One important feature of the view pane is the incorporation of the static code statistics and software complexity metrics described in Section 4.4.1. The statistics can be viewed to manually analyze and compare the complexity of the classes and methods within an application. Such information can be used to determine the level of obfuscation or to check for a watermark. They can also be used to sort the methods in a class or the classes in a package. Within the view pane sorting can be done based on the software complexity metrics or the occurrence of instruction opcodes.

At the application level, the viewing options include the various software complexity metrics, the inheritance graph, the call graph, and the application CFG. To illustrate an attack which could be performed using this information, consider a simple watermarking algorithm in which the watermark is embedded by adding a fake method. If this method is not called by another method in the application, this information will be visible using the call graph.

Within the application, the viewing options are further customized at the class and method levels. Class level information includes the constant pool, the fields, static statistics, and complexity metrics. For each method it is possible to view the
Figure 4.6: The view pane makes it possible to view the call graph of the application.

bytecode instructions, the CFG, the interference graph, static statistics, complexity metrics, and forward and backward static slices based on a selected instruction. Figures 4.6, 4.7, and 4.8 show three examples of the information the view pane provides at the application, class, and method levels.

4.4.3 Java Bytecode Comparison Tool

The most serious vulnerability in software fingerprinting is the collusive attack. This attack arises when an adversary obtains several copies of the software, each of which contains a different fingerprint. Since the different copies of the software function the same, the attacker can compare the programs and identify where they differ. These differences identify the location of the fingerprint. To simulate a collusive attack, SANDMARK includes a Java bytecode comparison tool, the Diff tool.

The Diff tool computes the similarity between two Java applications and produces a listing of method pairs with their associated similarity measure. The similarity results are color-coded so that differing code segments can easily be identified. The Diff tool includes six different algorithms and the results can be automatically
Figure 4.7: At the class level various static code statistics can be observed using the view pane.

filtered based on the following criteria:

- Filter out pairs of objects that have the same name.
- Filter out pairs of objects that appear to be identical.
- Compare all pairs of objects.
- Compare only objects with the same name.
- Show pairs of objects with similarity greater than or equal to a specified filter value.
- Ignore methods with fewer than $n$ instructions.

The two most basic comparison algorithms are the opcode and literal diffs. The opcode diff simply compares the opcodes of the instructions. It does not take into consideration the arguments of the instruction. Thus, `iload 4` and `iload 5` instructions would be considered identical. The similarity measure is based on the
Figure 4.8: The bytecode instructions for each method can be examined using the view pane.

The longest common subsequence (LCS) of opcodes between the two methods.

$$opcodeSim(A, B) = \frac{|LCS|}{\min(|A|, |B|)}$$

The literal diff is the same as the opcode diff except that the literal diff also considers the arguments of the instructions. Figure 4.9 demonstrates the Diff tool by computing the similarity between iterative and recursive factorial programs using the literal diff.

The Baker-Manber diff algorithm is based on the dup tool developed by Baker and Manber for detecting similarities in Java source code [21]. dup is used to search source code files for matching sections taking into consideration systematic transformations. The Baker-Manber diff algorithm is an improvement over the opcode and literal diff algorithms because it can take into consideration minimal code transformations such as local variable reordering.

The constant pool diff is similar to the literal diff. Instead of comparing instructions, the constant pool tables are compared. The similarity is then computed using
the LCS.

The final two comparison algorithms are based on control flow structure. The basic block diff computes similarity by identifying the number of matching basic blocks in the two methods. The method CFG diff uses a naive algorithm to find the maximum common subtree between two method CFGs. The similarity is computed based on the ratio between the size of the subtree and the size of the larger CFG.

4.4.4 Optimization

Optimizations are useful in reducing the performance impact which is often incurred through obfuscation and watermarking. However, since optimizations are implemented through semantics-preserving transformations, they can often be used to destroy watermarks. sandMark, therefore, provides access to the BLOAT bytecode optimizer [2]. Additionally, three of the included obfuscations double as optimizations. These are static and dynamic method inlining and local variable reassignment (register reallocation).
4.5 Discussion

It is our hope that the SANDMARK tool will prove useful to software protection researchers in fairly evaluating their algorithms, to potential software protection users who wish to evaluate different protection mechanisms, and to software developers who wish to protect their software from piracy, reverse engineering, or tampering by using software protection algorithms. The plugin architecture and extensive support code make it easy for researchers to add and evaluate their own protection algorithms. Through the combination of the various protection and evaluation techniques it is easy to combine algorithms, evaluate the effectiveness and performance overhead of the algorithms, and to simulate manual and automated attacks against the algorithms.
III

SOFTWARE WATERMARKING
CHAPTER 5

OVERVIEW

Software watermarking is one of the many techniques currently employed in the battle against software piracy. Instead of approaching the issue of piracy from the more difficult direction of prevention, the immediate goal of watermarking is to prove ownership of the intellectual property, while discouraging illegal use. One of the current flaws with legal deterrents is that the pirate is aware of the practical difficulties of enforcement. However, he may be less likely to redistribute a piece of software if he knows it can be traced back to him. Software watermarking provides a means to prove ownership of pirated software and in some cases even identify the source of the illegal distribution by permanently embedding a unique identifier in the data. This increases the perceived cost of piracy.

In this chapter we present a general overview of software watermarking and how it is used to aid in the protection of intellectual property. As motivation, we begin by outlining the practical applications of software watermarking. To provide a foundation for a solid understanding of the watermarking algorithms presented in future chapters, we formally present the different aspects of a software watermarking system. This is followed by an overview of the different characteristics used to classify a software watermarking algorithm. Finally, we define the threat model and the properties used to evaluate the strength of an algorithm.

5.1 Applications of Software Watermarking

Watermarking in general has many applications. Cox et al. [41] outline 7 categories: broadcast monitoring, owner identification, proof of ownership, transaction tracing, content authentication, copy control, and device control. Of these categories, we are particularly interested in the application of software watermarking to owner
identification, proof of ownership, and transaction tracking.

**Owner identification.** A watermark is embedded in the data which identifies the copyright holder. Owner identification is useful in instances where the visual copyright notice has been removed from the media. For example, if a module is removed from a program the visual copyright may no longer be associated with the code. The watermark will still allow someone to identify the copyright holder.

**Proof of ownership.** The embedded watermark identifies the original copyright holder. The mark is used in instances where the data was obtained legally but then redistributed illegally. The embedded watermark is a more robust proof of ownership than a visible copyright notice. The copyright notice can be altered, but if illegal redistribution is suspected the watermark can be extracted to prove ownership.

**Transaction tracking.** The embedded watermark records the transactions the particular copy of the data has taken part in. For example, the mark could be used to record the last legal sale of the media. If the copy was later illegally redistributed it would be possible to identify the last legal owner. A fingerprint is a specific type of transaction tracking since the mark encodes information about the legal recipient. A particularly useful application of transaction tracking is in the distribution of beta versions. The distribution is targeted at a select handful of users and designed to solicit feedback. Generally, the release of the beta version to a user is contingent on the software remaining confidential. Although this agreement is made, a leak frequently occurs. The embedding of a unique fingerprint tied to the user permits the company to take legal action against the identified party.
5.2 Definition

Software watermarking developed out of the principles of media watermarking, a technique used to protect images, audio, and video from illegal redistribution. Typically, media watermarking algorithms embed a unique identifier in a multimedia data object by introducing small errors which are undetectable by the human sensory system. Unfortunately, the software approach is slightly more difficult since the introduction of errors could adversely alter the functionality of the software. The preservation of functionality was formalized by Collberg and Thomborson [35]:

**Definition 4** (Watermarking Transformation). Let $P$ be a program, let $w(P)$ be a watermarking transformation of $P$, let $\text{dom}(P)$ be the set of input sequences accepted by $P$ and let $\text{out}(P, I)$ be the output of $P$ on input $I$. For each program $P$, for each input $I$ belonging to $\text{dom}(P)$, the following identities must be preserved:

- $\text{dom}(P) = \text{dom}(w(P))$ and
- $\text{out}(P, I) = \text{out}(w(P), I)$

The implication of this definition is that if the original program produces an error, the watermarked program will produce the same error. This definition is actually a bit more restrictive than necessary and can be relaxed:

**Definition 5** (Watermarking Transformation (relaxed)). Let $P$ be a program and $w(P) = P'$ be a watermarked transformation of $P$. If $P$ fails to terminate or terminates with an error, then $P'$ fails to terminate or terminates with an error. Otherwise, $P'$ must terminate and produce the same output as $P$.

While a watermarking transformation cannot alter the observable behavior of the program; the size, execution speed, and memory requirements can be altered. Thus, a watermarking transformation is similar to an optimizing transformation in that it must be semantics-preserving but can make modifications that affect the performance of the program.
The basic idea of software watermarking is to use a transformation to embed a message \( w \) (the “watermark”) into a program \( P \) such that \( w \) uniquely identifies the owner of \( P \) \((w \) is a copyright notice) or the purchaser of \( P \) \((w \) is a fingerprint). For watermarking to be useful in proof of ownership a software watermarking system must include the capabilities to embed and recognize a watermark. This leads us to the definition of a software watermarking system:

**Definition 6** (Software Watermarking System). Given a program \( P \) and a unique message \( w \), the most basic software watermarking system consists of two functions:

- \( \text{embed}(P, w, k) \rightarrow P' \) and
- \( \text{recognize}(P', k) \rightarrow w. \)

Using the secret key \( k \), the embed-function incorporates the watermark \( w \) into a program \( P \), yielding a new program \( P' \). The recognize-function uses the same key \( k \) to extract the watermark from a suspected pirated copy. The most important assumption about a watermarking system is that the original program, the message embedded in the software and the key used to embed and recognize the watermark are known only to the owner of the software, while the watermarking algorithm employed is known to the attacker as well as the owner.

From a legal perspective, to prove ownership, it is not sufficient to simply recover a mark from a program. It is also necessary to show that the watermark was intentionally embedded and that recognition is not by chance. Such requirements necessitate a formal definition of a software watermark \([35]\).

**Definition 7** (Software Watermark). Let \( W \) be a set of mathematical structures, and \( p \) a predicate such that \( \forall w \in W : p(w) \). We chose \( p \) and \( W \) such that the probability of \( p(x) \) for a random \( x \not\in W \) is small.

Choosing \( w \) such that \( w = pq \) where \( p \) and \( q \) are two large primes is one possible example of a strong watermark. Since factoring is a hard problem only the person who embedded such a watermark would be able to identify the factors \( p \) and \( q \).
5.3 Software Watermarking Classification

Each algorithm is categorized based on a set of characteristics. These include whether the code is analyzed in a static or dynamic manner, the type of recognizer used, the embedding technique, and the type of mark embedded. Each of these characteristics will be expanded on in the following sections.

5.3.1 Static or Dynamic Code Properties

Currently, all watermarking algorithms are either static, dynamic, or abstract. Whether the code is analyzed in a static or dynamic manner, or even a little of both, is an important feature of the watermarking system. Strictly static watermarking algorithms only make use of the features of an application that are available at compile-time, such as instruction sequences or the constant pool table in a Java application, to embed a watermark.

Definition 8 (Static Software Watermarking System). Given a program $P$, a watermark $w$, and a key $k$, a static software watermarking system consists of two functions:

- $\text{embed}(P, w, k) \rightarrow P'$
- $\text{recognize}(P', k) \rightarrow w$

On the other hand, a strictly dynamic watermarking algorithm relies on information gathered during the execution of the application to both embed and recognize the watermark.

Definition 9 (Dynamic Software Watermarking System). Given a program $P$, a watermark $w$, and a secret input sequence $I$, a dynamic software watermarking system consists of two functions:

- $\text{embed}(P, w, I) \rightarrow P'$
- $\text{recognize}(P', I) \rightarrow w$
In a dynamic watermarking system the secret input sequence serves as the secret key. As the program is executed with the input sequence it will enter a state which will allow for watermark recovery. For example, if the software we wish to watermark is a tic-tac-toe program, the secret input could be the sequence of mouse clicks that selects “X-O-X” on the diagonal. The use of a secret input sequence makes it possible to store the watermark in the program’s execution state rather than in the static code itself.

There are three different types of dynamic watermarks: easter egg watermarks, data structure watermarks, and execution trace watermarks [35]. The difference in these techniques is how the watermark is stored in the state and how it is recovered.

An easter egg watermark is a piece of code that can only be executed by a highly unusual input sequence. Once the input sequence has been entered the easter egg is displayed to the user. For example, in Adobe Acrobat Reader 4.0 select Help → About Plug-ins → Acrobat Forms and then hold Control+Alt+Shift while clicking on the credits button. This special input sequence will reveal the easter egg in Figure 5.1 that consists of three features: a dog bark, a credit button face that changes to say “woof”, and an Adobe emblem that becomes a dog paw [5]. The drawback to using easter eggs to embed a watermark is that once they have been discovered, simple debugging techniques make it easy to locate and remove the watermark.

A data structure watermark embeds the watermark in the state of the program (such as the global, heap, and stack data) it is executed with a particular input. Because the watermark is stored within the state of the program, recognition requires analysis of the dynamic behavior. The analysis can be gathered in a variety of ways. For instance, the code can be instrumented or a debugger can be used to observe and gather information about the execution behavior. This technique is far more stealthy than an easter egg watermark since no output is produced.

The third technique, execution trace watermarking, embeds the watermark within the trace of the application as it is executed with a particular input. This technique differs from the data structure watermark in that the watermark is em-
embedded in the application’s instructions or addresses instead of the application’s state such as the heap. To recognize the watermark, properties about the execution are monitored.

Abstract watermarking is a technique which relies on abstract interpretation to construct a watermarking system. The technique cannot be classified as strictly static or dynamic. A watermarking transformation which is based on abstract interpretation embeds the watermark in such a way that it can only be extracted by analyzing the concrete semantics of the code. The technique is static in that recognition does not require execution of the program. However, it can be considered dynamic since the watermark is hidden in the semantics of the program.

The number of static watermarking schemes outnumber both abstract and dynamic. This is due to the multitude of locations where the information can be hidden in an executable. Some of the current research is focused on which of the three techniques is the strongest. As of now, all that can be said is that the algorithms make use of different information to embed the watermark. Because of the different information used, dynamic algorithms are currently not useful in watermarking individual modules, only whole programs. This is in contrast to the flexibility of static algorithms. However, it is hypothesized that static algorithms are more vulnerable to attack.
5.3.2 Recognition Type

A watermark recognizer is categorized based on the information needed to identify the watermark. The ideal system uses a blind recognition procedure. A blind recognizer only requires access to the watermarked program and the secret key used during embedding to extract the watermark:

\[ \text{recognize}(P', k) \rightarrow w \]  \hspace{1cm} (5.1)

On the other hand, in addition to the watermarked program and secret key, an informed recognition procedure requires access to the unwatermarked program and/or the watermark. An informed recognizer can have one of the following signatures:

\[ \text{recognize}(P', P, k) \rightarrow w \]  \hspace{1cm} (5.2)

\[ \text{recognize}(P', P, k, w, \text{fuzz}) \rightarrow \mathbb{B} \]  \hspace{1cm} (5.3)

\[ \text{recognize}(P', k, w, \text{fuzz}) \rightarrow \mathbb{B} \]  \hspace{1cm} (5.4)

Recognizers (5.3) and (5.4) would return true if \( w \) exists in \( P' \) with probability fuzz, where fuzz is a number between 0 and 1. In each of the above recognizers the secret input sequence \( I \) can be substituted for the secret key \( k \) for dynamic algorithms.

5.3.3 Embedding Technique

To incorporate a watermark, a program has to be manipulated through a semantics-preserving transformation. Even though each embedding technique applies a unique transformation, the transformations can be categorized.

- The most naive approach relies on reordering or renaming to embed the mark. Statements or basic blocks of code can be reordered; the particular order chosen represents the watermark. Renaming can modify identifiers throughout the program or alter which particular registers are used.
• A transformation which alters the program’s semantics can be used for embedding. The watermark is encoded by inserting new (non-functional or never executed) code.

• A program’s statistical properties such as instruction frequencies can be manipulated to embed a watermark.

5.3.4 Embedded Mark Type

Software watermarking takes the approach of discouraging piracy through the attachment of an identifying mark. An authorship mark (AM) is a watermark in which the same mark is embedded in every copy of the program [79]. An AM is used to identify the author and is in essence a copyright notice. This type of mark can be used by a developer to prove ownership of pirated software. Additionally, the identifier is extremely helpful in proving that a module was illegally extracted from a competitor’s program and reused. For example, suppose a programmer from Company B steals a secret module from Company A’s software to decrease his own development time. If Company A can demonstrate that Company B’s software contains their watermark then company A can prove that Company B is illegally profiting from something that is not theirs. Unfortunately, an authorship mark only proves that Company B was using the secret and not that they were the ones that stole it.

On the other hand, a fingerprint mark (FM) is unique for each copy distributed and is normally used to identify the purchaser [79]. Through the use of a FM it is possible to trace the program back to the original purchaser. To illustrate, suppose Alice sells Bob a copy of her software. Before she gives Bob the copy she embeds his credit card number. When Alice obtains a copy of her software, which she believes is pirated, she uses the recognize function with her secret key to extract the watermark. Since the watermark is Bob’s credit card number, which uniquely identifies him, she can prove that Bob is the guilty pirate.

Both marks should be robust against tampering, however only the fingerprint
mark requires invisibility. In specific instances it may be desirable for the authorship
mark to be visible, e.g. the mark conveys a level of quality. In instances where the
mark is used in a potentially hostile environment to protect a secret or for tracing
piracy the invisibility may increase the strength of the mark.

5.4 Software Watermark Evaluation

In order for a software watermarking technique to be effective against software piracy
and copyright infringement it should be resilient against determined attempts at
discovery and removal. Prior to the development of SANDMARK very little work
had been done on evaluating the strength of software watermarking systems. Such
limited evaluation makes it difficult to adequately compare the various techniques.
In this section we outline a universal set of properties which facilitate the evaluation
and comparison of software watermarking schemes.

5.4.1 Threat Model

The strength of a software watermarking algorithm is evaluated based on a well-
defined threat model. The threat model describes tools and techniques generally
employed by an attacker. Such attacks are categorized as manual, automated, and
blended. In a manual attack the software is analyzed and modified by hand using
reverse engineering techniques. An automated attack is characterized by the use of
tools which automatically apply an attack such as the DeCSS script used to disable
DVD encryption. The most common form of attack, the blended attack, uses both
manual and automated techniques to disable the protection. For example, disas-
semblers, debuggers, and decompilers are often used by attackers to interactively
explore applications. Information gleaned from this analysis can be used to develop
automated tools for disabling the protection on all copies of the program.

Since an attacker has full control over the execution of the software, it is generally
believed that given “enough” time, effort, and/or resources, a sufficiently determined
attacker can completely break any software protection technique. With this in mind,
most techniques are designed to make the cost of attack as high as possible. This can be accomplished by making the attack costly to carry out or by requiring an attack which degrades the performance of the software to an unacceptable level. For examples, a software watermarking algorithm could be considered effective if the attack required to destroy the watermark also slows down the un-watermarked program such that it no longer has any economic value.

The threat model specific to software watermarking algorithms includes four attacks: additive, distortive, subtractive, and collusive. To illustrate the attacks consider Alice and Bob. Alice produces a program $P$ which contains her watermark $w$. She sells a copy to Bob not aware that he wants to illegally redistribute the program. In order for Bob to successfully redistribute $P$ he needs to destroy the watermark $w$.

- **In an additive attack** Bob embeds his own watermark $w'$ into Alice’s program $P$. By doing this Bob has made it difficult for Alice to prove ownership since she will have to show that her watermark was embedded prior to Bob’s.

- **In a distortive attack** Bob applies a series of semantics-preserving transformations to $P$ in an attempt to destroy $w$. For Bob’s attack to be successful the watermark must be unrecognizable while preserving program functionality.

- **In a subtractive attack** Bob analyzes the (disassembled/decompiled) program $P$ to identify the location of the watermark and to remove all or part of it. As with a distortive attack the original functionality must be preserved for the attack to be successful.

- **In a collusive attack** Bob obtains two differently fingerprinted programs $P_1$ and $P_2$. Because the programs only differ in their fingerprints, Bob compares the programs to identify the location of the fingerprints.
5.4.2 Evaluation Properties

Through the study of software watermarking algorithms using the SANDMARK system we have compiled the following properties which we believe aid in evaluating the strength of an algorithm [35, 49, 86].

Data-rate: The ratio of the number of watermark bits that can be embedded to the size of the program.

Overhead: The decrease in performance and/or the increase in program size. To evaluate both performance and size impact the SANDMARK system includes the benchmark suites SpecJVM and CaffeineMark.

Credibility: The probability of false positives or false negatives.

Stealth: The degree of similarity between the watermark and the original code. Stealth can be characterized as either statistical or visual undetectability. The degree of statistical similarity can be measured using the statistics module within SANDMARK. A quantitative measure of visual stealth has yet to be establish. However, the general level of visual stealth can be evaluated using the view pane.

Robustness: The ability to withstand the four attacks described in the threat model. Each of these attacks can be simulated within SANDMARK. The watermarking module can be used to simulate an additive attack. The obfuscation module can be used to simulate a distortive attack. The statistics module and the view pane can be used to simulate a subtractive attack. The diff tool can be used to simulate a collusive attack.

As with any software protection technique, the design of a software watermarking algorithm generally requires a trade-off between the various evaluation properties. For example, the embedding of a larger watermark could decrease the level of stealth and increase the overhead. Additionally, to increase the robustness it may be necessary to have a larger impact on incurred overhead. A thorough evaluation of
the known software watermarking techniques based on the above properties makes it possible for a developer to choose the appropriate algorithm for the protection requirements. In Chapter 7 we use these properties to provide a preliminary evaluation of the known software watermarking algorithms. In addition to the preliminary evaluation, we use the properties in Chapter 9 to conduct a thorough evaluation of the Branch-based software watermarking technique and assess its strength in relation to the previously proposed techniques.

5.4.3 Evaluation Framework

Chapters 7-9 contain evaluations of watermarking techniques based on the five evaluation properties. To evaluate the strength of a watermarking technique with respect to a particular property requires a variety of different experiments. In these experiments we use one more more Java application which vary in size and complexity. Because we wanted to establish an automated evaluation framework we chose applications which run without human interaction. This criteria is necessary since dynamic watermarking requires program execution. Table 5.1 provides brief descriptions of each of the applications and Table 5.2 lists some of their characteristics. All experiments were run using Sun’s JVM version 1.4.2 and Redhat Linux 8.0.

Prior to running our experiments we used SANDMARK to apply a null obfuscation to each test application. In this obfuscation the application is read in and then written to a new jar file without performing any transformations. This step is performed to create a baseline for our size and performance experiments which will eliminate any effects (positive or negative) that SANDMARK has on the code the original compiler did not. The application characteristics in Table 5.2 were obtained after the null obfuscation was applied.

The first experiment is used to evaluate the data-rate of a watermarking technique. In this experiment we encode random strings in the test applications decode, fft, illness, lu, machineSim, matrix, probe, puzzle, Java Grande, and CaffeineMark. For each application, we increase the size of the watermark by 1 byte until the
<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>Implements an algorithm for decoding encrypted messages using Shamir’s Secret Sharing scheme.</td>
</tr>
<tr>
<td>fft</td>
<td>Fast Fourier Transform of complex double precision data.</td>
</tr>
<tr>
<td>illness</td>
<td>Simulates the spread of an illness through a group of people.</td>
</tr>
<tr>
<td>lu</td>
<td>LU matrix factorization.</td>
</tr>
<tr>
<td>machineSim</td>
<td>Simulates a Von Neumann machine.</td>
</tr>
<tr>
<td>matrix</td>
<td>Matrix inverter.</td>
</tr>
<tr>
<td>probe</td>
<td>Unknown.</td>
</tr>
<tr>
<td>puzzle</td>
<td>Takes four files describing puzzle pieces and attempts to determine how they fit together.</td>
</tr>
<tr>
<td>Java Grande</td>
<td>Set of five microbenchmarks which perform alpha-beta pruned search, computational fluid dynamics, molecular, dynamics simulation, monte carlo simulation, and 3D ray tracing.</td>
</tr>
<tr>
<td>CaffeineMark</td>
<td>Contains several microbenchmarks that test the performance of integer and floating point arithmetic operations, loops, logical operators and method calls.</td>
</tr>
</tbody>
</table>

Table 5.1: Brief description of each of the applications used in evaluating the strength of the watermarking techniques.

program can no longer hold the watermark or we reach 512 bits. We classify the algorithms as having a low (0 - 127 bits), usable (128 - 255 bits), high (256 - 511 bits), or exceptional (≥ 512 bits) data-rate.

In the second experiment we use the same ten applications as in the first experiment to evaluate the size overhead. Each of the applications is watermarked using 32-, 64-, 128-, 256-, and 512-bit watermarks. We then looked at the change in size between the original and the watermarked version.

The third experiment is used to evaluate the performance overhead associated with each watermarking algorithm. In this experiment we watermark the benchmark suites CaffeineMark and Java Grande with 32, 64, 128, 256, and 512 bit watermarks if possible. Otherwise, we use the largest watermark the application will hold. The execution times reported are obtained through 5 runs. From these runs the geometric mean is computed. For each of the overhead experiments the algorithms are rated as having a low (0-19%), moderate (20-49%), high (50-99%), or extreme (≥ 100%) impact on performance.
Table 5.2: Characteristics of each of the Java applications used in evaluating the strength of the watermarking techniques. Method and total sizes are given in bytes.

<table>
<thead>
<tr>
<th>Application</th>
<th>total size</th>
<th>total classes</th>
<th>total methods</th>
<th>total fields</th>
<th>max method</th>
<th>avg method</th>
<th>min method</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>5728</td>
<td>4</td>
<td>20</td>
<td>10</td>
<td>1065</td>
<td>137</td>
<td>5</td>
</tr>
<tr>
<td>fft</td>
<td>3136</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>386</td>
<td>98</td>
<td>5</td>
</tr>
<tr>
<td>illness</td>
<td>13735</td>
<td>16</td>
<td>104</td>
<td>27</td>
<td>2220</td>
<td>60</td>
<td>1</td>
</tr>
<tr>
<td>lu</td>
<td>2744</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>271</td>
<td>107</td>
<td>5</td>
</tr>
<tr>
<td>machineSim</td>
<td>17751</td>
<td>12</td>
<td>110</td>
<td>13</td>
<td>1100</td>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>matrix</td>
<td>2399</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>541</td>
<td>105</td>
<td>9</td>
</tr>
<tr>
<td>probe</td>
<td>2699</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>388</td>
<td>127</td>
<td>5</td>
</tr>
<tr>
<td>puzzle</td>
<td>8627</td>
<td>3</td>
<td>20</td>
<td>10</td>
<td>860</td>
<td>273</td>
<td>5</td>
</tr>
<tr>
<td>Java Grande</td>
<td>198976</td>
<td>56</td>
<td>387</td>
<td>330</td>
<td>2984</td>
<td>81</td>
<td>1</td>
</tr>
<tr>
<td>CaffeineMark</td>
<td>9063</td>
<td>12</td>
<td>88</td>
<td>41</td>
<td>708</td>
<td>33</td>
<td>1</td>
</tr>
</tbody>
</table>

To evaluate the robustness of a watermarking technique against distortive attacks we use the 35 different obfuscations from SANDMARK. (A brief description of each obfuscation can be found in Appendix B.) The obfuscations perform a variety of code transformations which represent a reasonable sampling of possible distortive attacks. There is at least one obfuscation in each of the general categories: layout, data, and control flow obfuscation. Additionally, a distortive transformation used to attack a watermarked application will be made up of one or more of the following operations [87]:

**fold/flatten:** turn a $d$-dimensional construct into one with $d+1$ or $d-1$-dimensions;

**split/merge:** turn a compound construct $X$ into two constructs $\{a, b\}$ or two constructs $a$ and $b$ into a compound construct $X$;

**box/unbox:** add or remove a layer of abstraction;

**ref/deref:** add or remove a level of indirection;

**reorder:** swap two adjacent constructs;

**rename:** assign a new name to a labeled construct.

In this context, construct refers to any programming language object the transformation can alter. Each SANDMARK obfuscation performs one or more of these
operations.

The first part of the evaluation involves a theoretical consideration of the effects each obfuscating transformation could have on an application and the types of transformations a watermarked application is susceptible to. Based on this we identify the set of SANDMARK obfuscations which have the potential to destroy the watermark given the right combination of program, watermark value, and key value. Next, using each of the obfuscations individually, we automatically transform a test program into a semantically equivalent, but not identical program. In this experiment, we watermark the applications decode, fft, illness, lu, machineSim, matrix, probe, puzzle, and Java Grande with a 128 bit watermark. We then apply the obfuscations. For each obfuscation which successfully destroyed the watermark in the Java Grande application, we evaluate the performance impact associated with the obfuscation.

To evaluate the robustness against additive attacks we examine the attack from two perspectives. First, we consider the case when an application is double watermarked using the same watermarking technique but different watermark and key values. Then we look at the effects of applying a second watermark using a different watermarking technique. To perform this evaluation we not only consider the theoretical implications but we also perform an experimental evaluation using the watermarking algorithms contained in SANDMARK.
CHAPTER 6

NAÏVE WATERMARKING ALGORITHMS

The sophistication of software watermarking schemes has increased significantly since the mid 1990s. To illustrate this evolution, in this chapter we will present a few of the early naïve watermarking schemes. The limited usefulness of each of the algorithms is illustrated through an evaluation with respect to the properties outlined in Chapter 5.

6.1 Constant String

The most basic watermarking technique is to insert a constant string in the program. The string could be a copyright notice such as:

String watermark = "Copyright 2005 ABC Software Company"

This approach could also be used for fingerprinting by embedding the purchaser’s credit card number:

String fingerprint = "CC Number 0123 4567 8901 2345"

Extraction of the watermark using a blind recognizer is accomplished by searching for the identifier name assigned to the watermark string. In the above example we would search for the string ‘watermark’.

The strength of the algorithm lies in its high data-rate and credibility and low overhead. The statistical stealth depends on the original program. If the original did not contain character strings then the statistical stealth will be low. Additionally, the visual stealth is low because a visual scan of the program will reveal the constant string. The major weakness of the algorithm is its inability to withstand the four
attacks. The distortive attack with the least overhead would be to simply rename the program identifiers. Such an attack does not damage the watermark string but it does defeat the recognizer. Alternately, a dead code optimizer could be applied to the watermarked program. Since the constant string is never used it will be identified as dead code and removed. The low resilience to a subtractive attack is directly linked to the low visual stealth. On a Java classfile a subtractive attack can be performed by first using javap which will display the bytecode instructions. Once the watermark string location has been identified it can be removed using a bytecode editing tool. For a native executable the watermark can be removed using a disassembler, such as IDA Pro [6], and a hex editor.

The Constant String technique can also be extended so that the watermark is split into multiple pieces and embedded throughout the program. For example, suppose we split the watermark into \( n \) pieces, \( W = w_1 w_2 \ldots w_n \). For each \( w_i \) we add the constant string \( wm$s_i = w_i \) to the program. The watermark is recovered by identifying each \( wm$s_i \) and concatenating the values.

Splitting the watermark into multiple pieces is a technique commonly used in software watermarking algorithms to improve the strength of the algorithm. This enables the watermark to be distributed throughout the program thereby decreasing the probability that the watermark is altered when small changes are made to the application. Because this is an important aspect of the strength of an algorithm, the SandMark tool includes several different watermark splitting techniques. For example, one technique splits the watermark into \( k \) pieces \( \{ w_1, w_2, \ldots, w_k \} \) such that \( 0 \leq w_i \leq n \) and recovery of the pieces is order independent. The technique used to split the watermark relies on a 1-1 correspondence between a multiset \( S \) of size \( m \) (where \( S = \{ s_i : 0 \leq s_i \leq n \} \)) and combinations of size \( n \) chosen from \( m + n \) elements. Given this correspondence, the splitter enumerates combinations of \( n \) chosen from the \( m + n \) elements for some fixed \( n \).
Figure 6.1: Class after embedding the watermark \texttt{hello} using the \texttt{Identifier Encoding} algorithm.

### 6.2 \texttt{Identifier Encoding}

The \texttt{Identifier Encoding} algorithm embeds the watermark by attaching watermark pieces to the various identifiers in the program. The embedding is accomplished by splitting the watermark into \(n\) pieces, \(W = w_1w_2...w_n\). Each piece \(w_i\) is added to a named language construct which survives compilation. In Java this would be class, field, and method names. In C, external function names could be used. To implement this algorithm it is necessary to add a special character, e.g. \$\$, prior to adding the watermark piece to the identifier name. This particular detail enables blind recognition. During recognition each identifier name is examined. The watermark pieces are extracted from those containing the special character and assembled to form the watermark. To enable extraction of the pieces in any order, order independent splitting techniques can be employed to break the watermark into pieces. Figure 6.1 shows a class before and after embedding the watermark \texttt{hello} using the \texttt{Identifier Encoding} technique.

As with the \texttt{Constant String} technique, \texttt{Identifier Encoding} has limited usefulness due to its lack of resistance to attack. The algorithm does demonstrate high datarate and credibility and low overhead. However, the overall stealth is low due to the necessary introduction of the special character. Even though the particular character used does not need to be revealed, the fact that it is selected such that it does not normally appear in identifier names makes the algorithm susceptible to
statistical and visual analysis. The simplicity of the required analysis leaves the technique vulnerable to subtractive attacks. Additionally, a successful distortive attack can be performed which requires very little skill by the attacker and has little performance impact on the program. All traces of the watermark are eliminated by applying an obfuscation which renames the identifiers in the program.

6.3 Switch Encoding

The switch statement can be used in a variety of different ways to encode a watermark. The Bogus Switch algorithm embeds the watermark in the cases of a switch block. The first step in such an encoding generally involves breaking the watermark into several pieces, \( W = w_1w_2 \cdots w_n \). A new switch statement is added to the application with a single case for each watermark piece. To aid in reassembling the watermark, each case would typically contain information used to indicate the order of the pieces. The switch statement below is an example of how the watermark 654 could be embedded in Java source code:

```java
int a = 0;
switch(a) {
    case 16:
        break;
    case 25:
        break;
    case 34:
        break;
}
```

In this encoding, the first digit of a case label encodes the position of the watermark piece in the watermark, and the second digit encodes the watermark piece. By using such an encoding, the watermark can survive reordering of the cases.

As with the previous two schemes, the Bogus Switch watermark can be easily defeated, even if an attacker cannot find the specific switch statement encoding the watermark. For example, running an optimizer over the code which eliminates dead code would have the effect of removing the variable \( a \) and the entire switch block.
To thwart such an attack additional use of a would need to be added before, in, and after the switch block.

In a slightly more involved attack, an attacker could modify the switch statements and each of the case labels by modifying the value of a before the switch and undoing the modification before the next use. The transformation below illustrates this particular attack. When the recognizer is run on the attacked application the watermark 987 is detected.

```c
int a = 0;
switch(a) {
    case 16:
        break;
    case 25:
        break;
    case 34:
        break;
}
```

```c
int a = 0;
switch(a) {
    case 19:
        break;
    case 28:
        break;
    case 37:
        break;
}
```

In addition to the lack of resistance to attack, the Bogus Switch watermarking scheme has a greater impact on the program than the previous naïve techniques. To embed the watermark an entire switch statement is added to the program.

A second technique used to embed a watermark in a switch statement is to identify a switch whose cases can be reordered. The new order chosen represents the watermark. For example, to embed the watermark “5” in a switch statement that has the 3 cases 1,3 and 5, the 5th element in the natural order permutation of (1, 3, 5) is selected. Thus, we select (5, 1, 3) from the permutation set \{(1, 3, 5), (1, 5, 3), (3, 1, 5), (3, 5, 1), (5, 1, 3), (5, 3, 1)\}. Such an embedding yields the following transformation:

```c
switch(x){
    case 1: {...} 
    case 3: {...} 
    case 5: {...} 
}
```

```c
switch(x){
    case 5: {...} 
    case 1: {...} 
    case 3: {...} 
}
```
The Bogus Switch technique requires the insertion of additional code. The Switch Reorder technique improves on this weakness by simply using the code which already exists. Unfortunately, this aspect of the algorithm does constrain the size of watermark which can be embedded. The number and size of the switch statements dictates the watermarks which can be embedded. Thus, the Switch Reorder technique has the potential to have a low data-rate. Additionally, because the technique only rearranges code that already exists in the program the level of stealth is an improvement over the Bogus Switch encoding. The ease with which a watermark can be embedded using this technique makes it vulnerable to both additive and distortive attacks. A second watermark embedded using the same technique will again reorder the switch statements eliminating the original mark. Similarly, a distortive attack can be constructed which reorders all switch statements destroying the watermark.

6.4 Discussion

The naïve watermarking techniques presented in this chapter begin to illustrate the challenges associated with watermarking software. Those program features which are easy to manipulate for embedding purposes are also easy for an adversary to manipulate in an attack. Additionally, these techniques illustrate the penalties often incurred when one watermark technique attempts to improve on a weak property in another technique. For example, the Bogus Switch technique has a negative impact on the size of the program. The Switch Reorder technique eliminates this weakness. However, in doing so the size of the watermark is constrained. Such challenges and trade-offs are common to all software watermarking techniques. Through an evaluation of each of the techniques a software developer can identify the technique whose properties best fit the given application and requirements.
CHAPTER 7

PUBLISHED WATERMARKING TECHNIQUES

In this chapter we present a thorough treatment of the more advanced published software watermarking techniques. The techniques are presented chronologically within categories of static, dynamic, and abstract. Figure 7.1 classifies each of the algorithms based on the characteristics presented in Chapter 5. This chapter will serve as a basis for evaluating the novel Branch-based watermarking scheme presented in Chapter 9. Additionally, it will serve as a basis for other researchers studying the area of software watermarking. To the best of our knowledge, no other single publication exists which discusses, evaluates, and compares such an extensive collection of published software watermarking techniques.

7.1 Davidson-Myhrvold Algorithm

The Davidson-Myhrvold (DM) watermarking algorithm [42] embeds the watermark in the code segment of a program via a reordering of basic blocks in a function CFG. Based on the watermark value a unique permutation, called the placement order file, is generated. The length of the permutation dictates the number of continuous basic blocks selected for reordering. The selected blocks are then reordered according to the placement order file. Finally, to maintain the original functionality the blocks are relinked such that the proper control flow is preserved. This is accomplished by adding branch instructions to the end of those blocks that do not fall through to the appropriate blocks. The patent describing the algorithm does not provide specifics on watermark recovery. It simply states that the placement order can be recovered, if each block is unique, by identifying the blocks based on their bit patterns. In other words, the watermarked block sequence is compared with the original block sequence to recover the placement order file.
Figure 7.1: Classification of each the published software watermarking algorithms presented in this chapter.

To the best of our knowledge, the DM algorithm has been implemented and empirically evaluated on two separate occasions. The first implementation was performed by Hattanda and Ichikawa [52] for the MIPS platform. The second implementation was by Myles et al. [77] for the Java platform and was incorporated into the SandMark tool.

### 7.1.1 MIPS Version

Hattanda and Ichikawa present a very high level description of the DM algorithm for the MIPS platform which is missing many of the implementation details. Embedding the watermark involves a series of four steps. First, for each basic block in the original code segment, an MD5 hash is computed. If collisions occur between basic block hash values, NOP instructions are added. Second, an ordering is imposed on the basic blocks based on their hash values. Third, the basic blocks are reordered to embed a watermark. However, to preserve the entry address, the initial basic block is excluded from permutation. Finally, branch instructions are inserted to preserve the original functionality. To recover the watermark, the order of the original basic
blocks is compared to the watermarked order.

The watermark bits are encoded in the block permutation using the Partial Permutation Scheme [35]. In this scheme the \( n \) basic blocks are split into groups of size \( k \), where \( k \leq n \). Each group can encode \( \lceil \log_2 k! \rceil \) bits, thus, a code segment with \( n \) basic blocks, can encode \( \left( \left\lfloor \frac{n}{k} \right\rfloor \right) \left( \lceil \log_2 k! \rceil \right) \) total bits. For their implementation \( k = 6 \). This leads to each group encoding 8 bits of the watermark with an additional bit as an integrity check.

**Evaluation**

Hattanda and Ichikawa limit the evaluation of their implementation to data-rate and impact on performance and size. Ideally, the data-rate for \( n \) basic blocks would be \( \left\lfloor \frac{n}{k} \right\rfloor \). However, implementation details such as excluding the initial basic block prevent the use of certain basic blocks. Despite this fact, close to optimal data-rate is achieved on the 8 test files which range in size from 13 to 408 basic blocks.

To evaluate the impact on program size 100 watermarks were generated for each of the 8 test cases. The size increase ranged from 9\% to 24\%. Performance impact was evaluated using 100 runs of the benchmarks Dhrystone, Linpack, and Whetstone. Overall, performance of the watermarked benchmarks was 86\% to 102\% of the original.

### 7.1.2 Java Version

As part of the study of software-based techniques for software protection, the DM algorithm has been implemented and evaluated within the SANDMARK framework. Because the DM algorithm was originally described in a patent, many of the implementation details are missing. One such example is the technique for generating the placement order file. In the implementation we fill in the details making choices which maintain or improve the strength of the algorithm.
Embedding

In the SandMark implementation the watermark, an integer, is converted into a permutation using an algorithm described by Knuth [58]. The watermark is converted into a permutation of length \( t \) where \( 0 \leq w < t! \). The integer sequence \( S = (1, ..., t) \) is permuted in the following manner:

Algorithm \texttt{Permute}(S):

1. for \( i \leftarrow 2 \) to \( t \) do
2. \hspace{1em} \( s \leftarrow w \mod i \)
3. \hspace{1em} \( w \leftarrow \lceil \frac{w}{i} \rceil \)
4. \hspace{1em} swap(\( S_i \), \( S_{s+1} \))
5. return \( S \)

Using the \texttt{Permute} algorithm the watermark 7 would be converted into the placement order file (3, 4, 1, 2).

Based on the length of the permutation a set of function CFGs are selected as candidates for embedding. A function is considered a valid candidate if it meets both of the following conditions:

1. The CFG for that function has at least as many unique blocks as the size of the placement order file. A block is unique if and only if no other block in the graph contains the same sequence of instructions.

2. For languages which use exception handling, the function cannot contain an exception table. There are cases where exception handling imposes an ordering on basic blocks that is difficult or impossible to alter. For example, it is not possible to produce a basic block ordering \( \langle b_2, b_1, b_0 \rangle \) for this code segment:

\[
\begin{align*}
&b_0 \\
&\textbf{try} \{ \\
&\hspace{1em} b_1 \\
&\}\textbf{catch} (\text{Exception e}) \{ \\
&\hspace{1em} b_2 \\
&\}
\end{align*}
\]
From the set of candidate functions, a single function is selected to embed the watermark. The appropriate number of continuous basic blocks are selected and then reordered according to the placement order file. Finally the blocks are relinked to maintain the behavior of the original program by appending unconditional branch instructions to the blocks that do not fall through to the appropriate blocks. In summary, the embedding algorithm is as follows:

Algorithm Embed\(\) (program, watermark):
1. ordering ← generatePermutation\(\) (watermark);
2. cfg ← selectCFG\(\) (program);
3. blocks ← cfg.getBasicBlocks\(\);
4. if ordering[0] ≠ 0 then
5. insert unconditional branch to block 0;
6. for \(i ← 0\) to ordering.length do
7. block ← blocks[ordering[i]];
8. function.add\(\) (block);
9. if last inst can fall through and fall through is not next in ordering then
10. insert unconditional branch to original next block;
11. if ordering.length < blocks.length then
12. add instructions from remaining blocks;

Continuing with the example of a watermark value of 7, the Embed algorithm will yield the watermark transformation illustrated in Figure 7.2.

Recognition

Extraction of a watermark embedded with the DM algorithm requires an informed recognizer of the form recognize\(\) \((P', P, k) → w\). The watermark is recovered by comparing the basic blocks in the watermarked CFG with the basic blocks in the original CFG. During the comparison a mapping is created between the original block and the watermark block. From the mapping the placement order file is recovered.
Figure 7.2: The resulting CFG for placement ordering (3, 4, 1, 2).

**Algorithm getMapping**(origCFG, wmCFG):
1. oBlocks ← origCFG.getBasicBlocks();
2. wmBlocks ← wmCFG.getBasicBlocks();
3. mapping ← new int[oBlocks.length];
4. for i ← 1 to oBlocks.length do
   5.   block ← oBlocks[i];
   6.   index ← wmBlocks.getIndexOf(block);
   7.   mapping[i] ← index;
8. return mapping;

Applying the **getMapping** algorithm on the CFGs in Figure 7.2 would yield the mapping (3, 4, 1, 2, 5, 6). At this point in the recognition process the length of the placement order file is unknown. However, it can be inferred that is is either 4, 5, or 6. Each of the possible permutations are converted back to integers using the following algorithm [58]:

\[
\begin{align*}
\text{GOTO} &\quad \text{(3, 4, 1, 2),} \\
\text{origCFG} &\quad \text{[3,4,1,2]} \\
\text{wmCFG} &\quad \text{[3,4,1,2]} \\
\text{oBlocks} &\quad \text{[3,4,1,2]} \\
\text{wmBlocks} &\quad \text{[3,4,1,2]} \\
\text{mapping} &\quad \text{[3,4,1,2]} \\
\text{index} &\quad \text{[3,4,1,2]} \\
\text{block} &\quad \text{[3,4,1,2]} \\
\end{align*}
\]
Algorithm \texttt{permToInt}(S):
1. \( i \leftarrow S\.\text{length}; \)
2. \( w \leftarrow 0; \)
3. \textbf{while} \( i > 1 \) \textbf{do}
4. \( \text{maxVal} \leftarrow \max\{S_1, ..., S_i\}; \)
5. \( s \leftarrow \text{indexOf}(\text{maxVal}); \)
6. \( w \leftarrow i \ast w + s - 1; \)
7. \( \text{swap}(S_i, S_s); \)
8. \( i \leftarrow i - 1; \)
9. \textbf{return} \( w; \)

Applying the \texttt{permToInt} algorithm to the running example yields possible watermarks of 7, 103, and 703. To address the issue of recovering multiple watermarks the recognition algorithm could be of the form \texttt{recognize}(P', P, k, w, fuzz) \( \rightarrow \mathbb{B}. \) Alternatively, prior to embedding, prefix and postfix magic values can be added to the watermark. During recognition those recovered values which do not contain the magic values are discarded. This is the recognition procedure used in the \textsc{SandMark} implementation.

To further complicate the recognition process, the watermarked program may have been obfuscated or altered in some other way. Therefore, it is not known which method contains the watermark. To extract the watermark all possible pairs of original and watermark functions are compared. Putting all of these steps together yields the following recognition algorithm:
Algorithm Recognize\textit{(origP, wmP)}:

1. \textit{origMethods} $\leftarrow$ \textit{origP}.\textit{getMethods}();
2. \textit{wmMethods} $\leftarrow$ \textit{wmP}.\textit{getMethods}();
3. \textbf{for} $i \leftarrow 1$ \textbf{to} \textit{origMethods}.length \textbf{do}
   4. \textit{oMeth} $\leftarrow$ \textit{origMethods}[$i$];
   5. \textbf{for} $j \leftarrow 1$ \textbf{to} \textit{wmMethods}.length \textbf{do}
      6. \textit{wmMeth} $\leftarrow$ \textit{wmMethods}[$j$];
      7. \textit{origCFG} $\leftarrow$ \textit{oMeth}.\textit{getCFG}();
      8. \textit{wmCFG} $\leftarrow$ \textit{wmMeth}.\textit{getCFG}();
      9. \textit{mapping} $\leftarrow$ \textit{getMapping}($\textit{origCFG}$, \textit{wmCFG});
     10. \textbf{for} each possible permutation \textbf{do}
       11. \textit{wm} $\leftarrow$ \textit{PermToInt}(\textit{perm});
       12. \textbf{if} \textit{wm} contains magic characters \textbf{then}
           13. \textit{watermarks}.\textit{add}($\textit{wm}$);
       \textbf{else} discard \textit{wm};
     15. \textbf{return} \textit{watermarks}

Evaluation

The original evaluation of the DM algorithm for Java was conducted by Myles et al. [77]. In this section we provide a summary of those results with respect to the properties of credibility, data-rate, stealth, overhead, and robustness. In addition, we perform a further evaluation which will enable a comparison with the various software watermarking techniques presented in this chapter.

Credibility The DM algorithm provides a high level of credibility because it is guaranteed to recognize the watermark in a watermarked but otherwise unaltered program. The only flaw with respect to credibility is the possibility of recovering coincidental false watermarks. Even though the prefix and postfix magic values eliminate most false watermarks, there is still the possibility of recovering an ordering which is close, but not equal to the original ordering.
Figure 7.3: Maximum size watermark value that can be embedded in the ten test applications using the DM algorithm. The applications are organized in order of increasing size.

**Data-rate** The data-rate of the DM algorithm is linked to the size of the CFG for the largest method. A function with \( t \) basic blocks can contain an integer of size \( 0 \leq w < t! \). In the SANDMARK implementation the watermark value is initially viewed as a string. The string value is then encoded as an integer. Because of the encoding process the watermark string can be encoded as a very large integer. To examine the data-rate which could be expected in practice, we watermarked ten test applications with randomly generated watermark values. For each embedding, the length of the watermark value increased by one byte. Figure 7.3 shows that even for Java Grande only 18 bytes could be embedded. For an application of size 193298 bytes, this is a very low data-rate. Despite the low data-rate, 18 bytes is sufficient for many types of watermarks. For example, suppose we wish to embed a 16-digit credit card number. Two digits can be stored in one byte, thus 18 bytes is sufficient.

**Stealth** The stealth of the DM algorithm suffers due to the extreme changes made to a very localized piece of code. Since the basic blocks are rearranged, the function will contain many unconditional branch instructions. When applied to a Java program the unconditional branches will be `goto` instructions, which occur infrequently. Based on the analysis by Collberg et al. [34] only 1.9\% of instructions are
<table>
<thead>
<tr>
<th>Program</th>
<th>Program Ratio</th>
<th>Watermarked Method Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>0.03617</td>
<td>0.14176</td>
</tr>
<tr>
<td>fft</td>
<td>0.04019</td>
<td>0.16000</td>
</tr>
<tr>
<td>illness</td>
<td>0.02901</td>
<td>0.22500</td>
</tr>
<tr>
<td>lu</td>
<td>0.08407</td>
<td>0.15591</td>
</tr>
<tr>
<td>machineSim</td>
<td>0.03908</td>
<td>0.18709</td>
</tr>
<tr>
<td>matrix</td>
<td>0.08376</td>
<td>0.13175</td>
</tr>
<tr>
<td>probe</td>
<td>0.05543</td>
<td>0.28126</td>
</tr>
<tr>
<td>puzzle</td>
<td>0.03896</td>
<td>0.08793</td>
</tr>
</tbody>
</table>

Table 7.1: The ratio of goto instructions in the application compared to the ratio of goto instructions in the watermarked method.

goto instructions. Of the 10 applications we tested when the functions are sorted by the ratio of goto instructions within the functions, the watermarked functions was always ranked first. This property can be leveraged by an attacker to locate the watermarked function.

In the previous evaluation [77], the ratio of goto instructions to the total number of instructions in the program as well as the ratio of goto instructions to total number of instructions in the watermark method was calculated for eight of the test applications. Table 7.1 shows that the watermarked ratios are significantly greater than the overall program ratios. Such data further illustrates the lack of stealth exhibited by the DM algorithm.

**Overhead** The performance and size overhead incurred due to the DM watermarking algorithm is negligible. To embed the watermark the only additional code added are the unconditional branch instructions. Because the branch instructions are only needed when the natural fall through is disrupted, at most $n$ branches will be added for a permutation of size $n$.

Instead of adding to the method, the DM algorithm manipulates the method to embed the watermark. Because of this we were unable to embed a watermark of size 256 or 512 bits in any of the test applications. In fact, we were unable to embed even a 32-bit watermark in probe. Figure 7.4 illustrates that for most of
the applications the watermarked version is smaller than the original. For the two applications, *illness* and *CaffeineMark*, in which a size increase was reported, the increase was less than 1%.

Additionally, the execution time of the newly inserted branches is small. Embedding 32-, 64-, and 128-bit watermarks in the *Java Grande* benchmark application yields a slowdown of 0.6%, 1.7%, and 0.9% respectively. Embedding a 32-bit watermark in the *CaffeineMark* application resulted in no observable overall slowdown. Table 7.2 shows the results from watermarking the *CaffeineMark* application.

**Robustness** The lack of stealth leaves the DM algorithm highly vulnerable to a subtractive attack. To perform this attack the adversary can leverage the fact that the watermarked method contains an unusual number of unconditional branch instructions. Once the watermarked method is located, the attacker can simply rearrange the basic blocks to remove the watermark.

The DM algorithm is also highly susceptible to distortive attacks. Any modification which acts on all functions of the program is likely to erase the watermark. Also, because the recognition process requires a literal comparison between the instructions in the basic blocks, any modification to the instructions will result in watermark damage. If we consider the 35 *SandMark* obfuscations, the DM algo-
<table>
<thead>
<tr>
<th>Category</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sieve</td>
<td>-0.2%</td>
</tr>
<tr>
<td>Loop</td>
<td>-1%</td>
</tr>
<tr>
<td>Logic</td>
<td>0.2%</td>
</tr>
<tr>
<td>String</td>
<td>0.3%</td>
</tr>
<tr>
<td>Float</td>
<td>0.4%</td>
</tr>
<tr>
<td>Method</td>
<td>-0.1%</td>
</tr>
<tr>
<td>Overall</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

Table 7.2: CaffeineMark scores before and after embedding a 32 bit watermark using the DM algorithm.

algorithm, in theory, is susceptible to 31 of them. The four which it can withstand are class splitting, constant pool reordering, false refactoring, and overloading names.

Using the nine test applications, 20 of the SandMark obfuscations were able to destroy the embedded watermark in one or more applications. Results from applying the obfuscations to the nine applications are shown in Table 7.3. Nine of the obfuscations successfully destroyed the watermark embedded in the Java Grande benchmark. These include array splitting, bludgeoning signatures, inserting exception branches, inserting opaque predicates, merging local integers, promoting primitive registers, promoting primitive types, reordering instructions, and splitting classes. For this particular benchmark, the watermark can be destroyed with no performance implications. There was no observable slowdown when the watermark was destroyed by the reorder instructions obfuscation and only a 2% slowdown with merge local integers. Additionally, the obfuscations which insert opaque predicates, bludgeon signatures and split classes each resulted in less than a 20% slowdown. The remaining obfuscations had a more significant impact which ranged from 128% for promote primitive registers to 4014% for exception branches. Note that the obfuscations were applied uniformly throughout the application. If the attacker has some knowledge as to the location of the watermark, the obfuscation could be applied to a localized section of code. This would have the effect of destroying the watermark
while minimizing the impact on performance. Such attacks would be useful if it was necessary to apply one of the more performance degrading obfuscations.

Due to the fragility of the DM algorithm, an attacker could essentially replace the original proof of ownership with his own through an additive attack. Any watermarking algorithm which embeds across the entire program has the potential to damage the DM watermark. Those algorithms which embed in a single method are less likely to damage the watermark, but will still cast doubt on the original ownership.

Of the 11 watermarking algorithms presented in this chapter, 10 of the algorithms have the potential to destroy the original DM watermark. This includes the DM algorithm itself. For smaller applications, fewer candidate methods exist which can be manipulated to incorporate the watermark. Therefore, even if a different key and watermark value are used, the second watermark may be embedded in the method holding the original watermark. This was found to be true for the test applications fft, l1u, matrix, and probe. The Graph Coloring-Based algorithm is the only algorithm which will not destroy the original watermark, but it will cast doubt on the original ownership due to the presence of an additional watermark.

The DM algorithm embeds the entire watermark in a single method. Therefore, if an attacker obtains multiple differently watermarked applications, it is possible to perform a collusive attack. In the previous evaluation, such an attack was simulated by creating two differently watermarked versions of a tic-tac-toe application. Using the bytecode comparison tool, a basic block level comparison identified a method in each application as 100% similar, but containing a different block ordering. This information could lead an attacker to conclude that the watermark is embedded in the identified method.

7.2 Moskowitz-Cooperman Algorithm

Moskowitz and Cooperman [70] proposed a technique for watermarking computer applications through the use of digital watermarking. The technique relies on iden-
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array folder</td>
<td>✴️</td>
</tr>
<tr>
<td>Array splitter</td>
<td>✴️</td>
</tr>
<tr>
<td>Block marker</td>
<td>✴️</td>
</tr>
<tr>
<td>Bludgeon signatures</td>
<td>✴️</td>
</tr>
<tr>
<td>Boolean splitter</td>
<td>✴️</td>
</tr>
<tr>
<td>Branch inverter</td>
<td>✴️</td>
</tr>
<tr>
<td>Buggy code</td>
<td>✴️</td>
</tr>
<tr>
<td>Class splitter</td>
<td>✴️</td>
</tr>
<tr>
<td>Constant pool reorderer</td>
<td>✴️</td>
</tr>
<tr>
<td>Dublicate registers</td>
<td>✴️</td>
</tr>
<tr>
<td>Dynamic inliner</td>
<td>✴️</td>
</tr>
<tr>
<td>Exception branches</td>
<td>✴️</td>
</tr>
<tr>
<td>False refactor</td>
<td>✴️</td>
</tr>
<tr>
<td>Field assignment</td>
<td>✴️</td>
</tr>
<tr>
<td>Inliner</td>
<td>✴️</td>
</tr>
<tr>
<td>Insert opaque predicates</td>
<td>✴️</td>
</tr>
<tr>
<td>Interleave methods</td>
<td>✴️</td>
</tr>
<tr>
<td>Irreducibility</td>
<td>✴️</td>
</tr>
<tr>
<td>Merge local integers</td>
<td>✴️</td>
</tr>
<tr>
<td>Method merger</td>
<td>✴️</td>
</tr>
<tr>
<td>Objectify</td>
<td>✴️</td>
</tr>
<tr>
<td>Opaque branch insertion</td>
<td>✴️</td>
</tr>
<tr>
<td>Overload names</td>
<td>✴️</td>
</tr>
<tr>
<td>Parameter aliases</td>
<td>✴️</td>
</tr>
<tr>
<td>Promote primitive registers</td>
<td>✴️</td>
</tr>
<tr>
<td>Promote primitive types</td>
<td>✴️</td>
</tr>
<tr>
<td>Publicizer</td>
<td>✴️</td>
</tr>
<tr>
<td>Random dead code</td>
<td>✴️</td>
</tr>
<tr>
<td>Rename registers</td>
<td>✴️</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>✴️</td>
</tr>
<tr>
<td>Reorder parameters</td>
<td>✴️</td>
</tr>
<tr>
<td>Split classes</td>
<td>✴️</td>
</tr>
<tr>
<td>Static method bodies</td>
<td>✴️</td>
</tr>
<tr>
<td>String encoder</td>
<td>✴️</td>
</tr>
<tr>
<td>Variable reassigner</td>
<td>✴️</td>
</tr>
</tbody>
</table>

✴️ : watermark found in all test applications
✴️️ : watermark destroyed in at least one test application

Table 7.3: Results of applying the SANDMARK obfuscations on nine of the test applications which were watermarked using the DM algorithm.
tifying sections of the executable code which are considered essential to the proper functionality of the application. In this context, “essential” is defined to apply to those code sections which are executed reasonably often. Based on this, it is best not to choose sections of code which are optional or plug-ins because attackers can create patches to work around these sections. Along with the watermark, the identified sections of code are then embedded in image or audio data contained in the program through digital watermarking. Proper execution of the watermarked program relies on the extraction of the code sections from the data resources, thus, embedding the code sections serves as a method of tamperproofing the watermark.

7.2.1 Embedding

During embedding one or more of the chosen essential sections of code are encoded in one or more cover messages contained in the application through a digital watermarking process. The cover message used for the encoding could be image or audio samples which are important to the functionality of the application. Once encoded, the chosen code sections no longer appear in the object code in their original form, instead execution of the code segments requires that they be decoded from the cover message.

Extraction of the code segment, the digital watermark, during program execution requires the use of a key. The key is linked to the license information for the application. This is accomplished in one of two ways. Either the key can be generated from the license code or the key can be encrypted with a derivative of the license code and stored as a data resource. This technique allows the application to be copied but proper execution requires the inclusion of all licensing code.
Algorithm Embed($program, watermark$):
1. for each essential code section $code$ do
2. \hspace{1em} $cover \leftarrow getCoverMessage();$
3. \hspace{1em} digitalEmbed($cover, watermark, code$);
4. \hspace{1em} removeCode($code$);
5. for each reference to $code$ do
6. \hspace{2em} remove reference;
7. \hspace{2em} insert ‘$c = decode(cover, key)$’;
8. \hspace{2em} insert ‘execute($c$)’;

The Moskowitz-Cooperman technique has the advantage that if the watermarked data is attacked in an attempt to remove the license or copyright information (for example, by using the image transformations in the StirMark benchmark suite [82]), the code section is also likely to be damaged. This could result in non-functioning software. Figure 7.5 illustrates how the application is transformed in the watermarking process.

7.2.2 Recognition

Recovery of a watermark embedded using the Moskowitz-Cooperman algorithm uses a blind recognizer. The watermark is recovered by identifying the cover messages used during embedding. The recognition procedure for the digital watermarking technique is then applied to the cover messages.

Algorithm Recognize($program$):
1. for each cover message $message$ do
2. \hspace{1em} $wm \leftarrow digitalRecognize(message)$;
3. \hspace{1em} watermarks.add($wm$);
4. return $watermarks$;

The strength of the Moskowitz-Cooperman algorithm lies in the incorporation of tamperproofing and digital watermarking, not in a clever code modification. Because digital watermarking techniques have been more thoroughly evaluated than
```java
class Example {
    void essential () {...}
    void main(String [] argv) {
        Example e = new Example();
        e.essential();
    }
}
```

(a) Original application.  

```java
class Example {
    void main(String [] argv) {
        Example e = new Example();
        Picture p = getPicture(key);
        Code c = decode(P);
        execute(c);
    }
}
```

(c) Transformed application.

Figure 7.5: Illustration of how an application is transformed using the Moskowitz-Cooperman watermarking process. The transformation is illustrated using Java source code for readability, however, the actual process would be performed on the bytecode.
software watermarking techniques, there is a better understanding of the strengths and weaknesses. This knowledge can then be used to choose a digital watermarking algorithm which provides the appropriate level of protection.

7.2.3 Evaluation

In this section we provide a complete evaluation of the Moskowitz-Cooperman algorithm using the SANDMARK Tool and the evaluation framework described in Chapter 5.

Credibility The Moskowitz-Cooperman algorithm uses a blind recognizer which extracts the watermark value, as opposed to indicating the probability of the watermark’s presence. In general, this type of recognition procedure yields a high level of credibility. Such is true for the Moskowitz-Cooperman algorithm, which is guaranteed to recognize the watermark in a watermarked, but otherwise unaltered program.

Data-rate The data-rate of the Moskowitz-Cooperman algorithm is linked to the data-rate of the image watermarking technique chosen. The test applications used in the evaluation do not include an image which could be used in watermarking. To overcome this difficulty we added the image in Figure 7.5(b), which adds an additional 356701 bytes to each application. Using the modified applications and the SANDMARK implementation we were able to embed watermark strings of varying lengths up to the maximum length tried of 64 bytes. Overall, the algorithm has the possibility of a high data-rate if a high data-rate image watermarking algorithm is used.

Stealth The stealth of the Moskowitz-Cooperman algorithm suffers mainly from the need to extract a section of code from an image in the program. Such an activity is not common in most applications and could easily alert an attacker. Additionally, not all applications include an image which can be used in the watermarking process.
Adding an unrelated image to the application can lead to decreased stealth.

**Overhead** The size impact incurred due to the SandMark implementation of the Moskowitz-Cooperman algorithm is quite significant. In measuring this impact we did not take into consideration the size of the image we added to even make watermarking possible, e.g. our baseline measure was the size of the application including the image. The majority of the size increase can be attributed to the inclusion of a customized class loader. The class loader is the portion of the watermarked application which extracts the embedded code from the image and loads it into memory so that it can be executed by the JVM. From Figure 7.6 it can be seen that for each test application, as the size of the watermark increases the size increase remains virtually constant. The impact on size was least significant for Java Grande which is the largest application. The increase remained constant at 46%. The other nine applications incurred approximately a 70% increase in size after watermarking.

Despite the need for a customized class loader, the SandMark implementation of the Moskowitz-Cooperman algorithm has little impact on performance. Embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the Java Grande application lead to a slowdown of less than 0.5% in each case. For CaffeineMark, the most
<table>
<thead>
<tr>
<th>Category</th>
<th>32-bit</th>
<th>64-bit</th>
<th>128-bit</th>
<th>256-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sieve</td>
<td>0.2%</td>
<td>0.3%</td>
<td>3.0%</td>
<td>-0.2%</td>
<td>0.3%</td>
</tr>
<tr>
<td>Loop</td>
<td>-14.3%</td>
<td>2.7%</td>
<td>1.7%</td>
<td>1.1%</td>
<td>3.1%</td>
</tr>
<tr>
<td>Logic</td>
<td>0.7%</td>
<td>0.8%</td>
<td>1.0%</td>
<td>0.6%</td>
<td>0.7%</td>
</tr>
<tr>
<td>String</td>
<td>1.5%</td>
<td>2.8%</td>
<td>1.5%</td>
<td>1.2%</td>
<td>1.7%</td>
</tr>
<tr>
<td>Float</td>
<td>-5.3%</td>
<td>2.3%</td>
<td>-1.9%</td>
<td>-1.5%</td>
<td>-2.8%</td>
</tr>
<tr>
<td>Method</td>
<td>0%</td>
<td>0.9%</td>
<td>1.4%</td>
<td>1.1%</td>
<td>0.9%</td>
</tr>
<tr>
<td>Overall</td>
<td>-4.2%</td>
<td>1.7%</td>
<td>1.2%</td>
<td>0.4%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

Table 7.4: *CaffeineMark* scores before and after embedding 32-, 64-, 128-, 256-, and 512-bit watermarks using the *Moskowitz-Cooperman* algorithm.

Significant slowdown observed was still less than 2% and for the 32-bit watermark a performance improvement was recorded. Table 7.4 shows the results from embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the *CaffeineMark* application. In the *SandMark* implementation of the *Moskowitz-Cooperman* algorithm tamper-proofing is accomplished by embedding an entire class in an image file. Currently the class selected is the smallest class in the application. This obviously has implications for any performance analysis. We would expect the encoding of a larger class to have more of an impact on the overall performance of the watermarked application.

**Robustness** The low stealth of the *Moskowitz-Cooperman* algorithm provides an attacker with leverage in launching a subtractive attack. However, such an attack requires slightly more sophistication than for the *DM* algorithm. The extraction of code from an image is unusual behavior for many programs. By noticing the unusual behavior, an attacker knows exactly where to begin monitoring the program’s execution. Once the code segment has been extracted the attacker can capture the instruction sequence and patch the necessary locations throughout the program. This eliminates the tamperproofing mechanism making it possible to remove the watermark or possibly the entire image.
Because the watermark and essential code segments are embedded in images, the Moskowitz-Cooperman algorithm is not vulnerable to the conventional software watermarking distortive attacks. Because of this it is not vulnerable to any of the 35 obfuscations contained in SANDMARK. Additionally, because of the added tamperproofing feature it is also resistant to automated image transformations. Once the essential code segments have been extracted from the images and the program patched, using an attack such as the one described above, the watermark will be vulnerable to distortive attacks such as those contained in the StirMark benchmark.

As with the distortive attack, the tamperproofing mechanism enables the Moskowitz-Cooper to resist a double watermarking attack in which watermarked images are rewatermarked. However, the algorithm is not resistant to an additive attack in which a different watermarking technique is used to embed the watermark in the program. Additionally, the technique is not resistant to a double watermarking attack in which different images are used to embed the watermark.

As with the distortive attack, traditional software watermarking collusive attacks will not be successful in locating the watermark. Instead, the collusive attack will have to focus on the images in the application using image based techniques.

7.3 Graph Coloring-Based Watermarking

Graph Coloring-Based watermarking is a constraint-based static watermarking algorithm first proposed by Qu and Potkonjak [85, 86]. The idea behind constraint-based algorithms is to start with an optimization problem (usually an NP-complete problem such as graph coloring, scheduling, etc.) that is used to represent the original intellectual property. Then new constraints (representing the watermark) are added to this problem. Solving this new problem will yield a solution that embeds the watermark. Constraint-based watermarking is an especially useful technique for fingerprinting since it is generally costly to generate even a single solution to an optimization problem.

In the case of Graph Coloring-Based watermarking, the constraints are applied
to a method’s register interference graph so that the watermark can be encoded in
the register allocation of a program. (For an explanation of graph coloring based
register allocation see Chapter 3.) During the embedding phase a technique called
edge-adding is used to encode the watermark bits. Edges are added between chosen
vertices in a graph based on the value of the message. Once the vertices are
connected, coloring the graph will force the vertices to be colored with different
colors.

In this section we introduce the original graph coloring-based algorithm as it
was presented by Qu and Potkonjak. Through implementing and analyzing the
algorithm we developed a thorough understanding of its strengths and weaknesses.
Based on this understanding we made improvements to the algorithm which were
previously presented [73] and will be discussed in Chapter 8.

7.3.1 Embedding

Given a graph $G = (V, E)$ and a watermark $W$. Impose an ordering on the vertices
$V = (v_0, v_1, ..., v_{n-1})$ and convert $W$ to a binary representation, $W = w_0, w_1, ..., w_m$.
Using the embedding technique proposed by Qu and Potkonjak, $W$ is embedded in
$G$ as follows:

**Algorithm Embed**(function, watermark):
1. $G(V, E) \leftarrow$ interference graph for function;
2. $V \leftarrow \text{order}(V)$;
3. $G'(V', E') \leftarrow$ copy of $G(V, E)$;
4. for each $w_i \in W$ do
5. \hspace{1em} choose $v_i, v_{i_1}$ such that $(v_i, v_{i_1}), (v_i, v_{i_2}) \notin E$ and $i_2 > i_1 > i \pmod{n}$ and
\hspace{1em} $(v_i, v_j) \in E \forall i < j < i_1, i_1 < j < i_2 \pmod{n}$;
6. \hspace{1em} if $w_i == 0$ then add edge $(v_i, v_{i_1})$ to $E'$;
7. \hspace{1em} else add edge $(v_i, v_{i_2})$ to $E'$;
8. performRegisterAllocation($G'$);
Once the edges are added to the graph, the watermark bits are embedded in the program by using the watermark graph $G'$ in the register allocation. The new edges will force many of the vertices to be assigned to different registers than those used in the original allocation.

7.3.2 Recognition

The message is recognized as follows: Given a graph $G(V, E)$, for each pair of vertices $(v_i, v_j), j > i(mod n)$, which are not connected by an edge and are different colors, one bit of the message can be obtained. The bit extraction is done by examining how many vertices occur between $v_i$ and $v_j$ which are not connected to $v_i$. Let $V'$ be the set of vertices $v_k$ where $i < k < j$ and $(v_i, v_k) \notin E$. There are three cases to consider:

Case 1: If $|V'| == 0$ a 0 bit is extracted. This case is illustrated in Fig. 7.7(a) where solid edges represent edges in the original graph and dashed edges indicate an edge that was added during embedding. Suppose recognition is performed on the vertices $v_i = V1$ and $v_j = V3$. Since $V2$ is the only vertex between vertices $V1$ and $V3$ and is connected to vertex $V1$, a 0 bit is found.

Case 2: If $|V'| == 1$ then the hidden bit is 1. This case is illustrated using $v_i = V1$ and $v_j = V3$ in Fig. 7.7(b). Since vertex $V2$ lies between vertices $V1$ and $V3$ and is not connected to vertex $V1$ a 1 bit is recognized.

Case 3: If $|V'| \geq 2$ then reverse the order of the two vertices and repeat the process. To understand how this case could arise consider the graph in Fig. 7.7(b). Assume recognition is being performed on the vertices $v_i = V1$ and $v_j = 4$. Since there are two vertices between $V1$ and $V4$ that are not connected to $V1$, it is not possible that $V4$ was chosen as either $v_{i_1}$ or $v_{i_2}$ during embedding. Instead it must be that $V4$ was chosen as $v_i$ and $V1$ was chosen as $v_{i_1}$ which means that a 0 was embedded.

This yields the following recognition algorithm:
(a) Using case 1 a 0 bit can be extracted.  
(b) Using case 2 a 1 bit can be extracted and using case 3 a 0 bit can be extracted.

Figure 7.7: Illustration of the three recognition cases.

Algorithm Recognize(function):
1. $G(V,E) \leftarrow$ interference graph for function;
2. $V \leftarrow \text{order}(V)$;
3. for each $v_i, v_j \in V$ where $j > i \pmod{n}$ and $\text{color}(v_i) \neq \text{color}(v_j)$ and $(v_i,v_j) \notin E$ do
4. \hspace{1em} $\text{numVertices} \leftarrow 0$;
5. \hspace{1em} for each $v_k \in V$ where $i < k < j$ and $(v_i,v_k) \notin E$ do
6. \hspace{2em} $\text{numVertices}++$;
7. \hspace{1em} if $\text{numVertices} == 0$ then
8. \hspace{1.5em} $w_i = 0$;
9. \hspace{1em} else if $\text{numVertices} == 1$
10. \hspace{1.5em} $w_i = 1$;
11. \hspace{1em} else
12. \hspace{2em} swap($v_i$, $v_j$);
13. \hspace{2em} goto line 4;
14. \hspace{1em} return $W$;

7.3.3 Evaluation

The Graph Coloring-Based watermarking algorithm was originally evaluated by Myles and Collberg [73]. However, the evaluation results mainly focused on the proposed
algorithmic extensions. The extended algorithm, which is presented in Chapter 8, was developed in response to shortcomings identified while evaluating the Graph Coloring-Based algorithm. In the paper a discussion of the shortcomings is presented, but the completed evaluation is not. In this section we present a thorough evaluation of the original Graph Coloring-Based watermarking algorithm.

**Credibility** The recognition procedure used by the Graph Coloring-Based watermarking algorithm is a blind recognizer which extracts the watermark value. This type of recognition generally leads to a very high level of credibility. Unfortunately, when the Graph Coloring-Based algorithm is applied to software watermarking, it is not always possible to extract the correct watermark using the proposed recognition algorithm. The specifics as to why recognition fails are presented in Chapter 8.

To provide some quantitative measure of the level of credibility we watermarked the Java Grande and CaffeineMark benchmark applications with 10 random watermark strings of varying lengths. In each case the recognition algorithm failed to recover the correct watermark.

**Data-rate** The data-rate for the Graph Coloring-Based watermarking algorithm is upper bounded by the number of vertices in the interference graph. Based strictly on the embedding algorithm proposed by Qu and Potkonjak it appears that a single bit can be embedded for each vertex. Through careful inspection of the algorithm it is clear that it may not be possible to embed a bit for each vertex. In fact, in seven of the ten test applications we were unable to embed even a single byte. For probe we were able to embed 1 byte, for Java Grande 16 bytes, and CaffeineMark was able to hold a watermark of size 2 bytes. This is a very low data-rate for applications of size 2699, 198976, and 9963 bytes respectively.

**Stealth** A watermark embedded using the Graph Coloring-Based algorithm is extremely hard to detect. In the SANDMARK implementation no additional code is added to a watermarked method. Instead, the local variable number referenced by
the load and store instructions is altered. Extensive analysis would be required to discover that the local variable assignment is actually artificially imposed. Most likely such analysis would only tell an attacker that a more efficient local variable assignment could be used. In fact, such a conclusion could be drawn about most Java applications compiled using the javac or jikes compilers. From this we conclude that the Graph Coloring-Based watermarking algorithm demonstrates a high degree of stealth.

**Overhead** The Graph Coloring-Based watermarking algorithm introduces no additional code and makes only minimal code alterations. Because of this the impact on performance and size is negligible. Using the Java Grande benchmark application and watermarks of size 32 and 64 bits there was virtually no change in application size or performance.

**Robustness** Graph Coloring-Based watermarking is susceptible to a variety of attacks when applied to register allocation, with the simplest being the decompile/recompile attack. Since this algorithm embeds the watermark in the register allocation of the program, decompiling then recompiling the program is likely to reassign the registers. The decompile/recompile attack is a form of a subtractive attack.

The register allocation which encodes the watermark bits is artificially imposed and no code is added to the function to create an actual interference in the graph. Because of this the watermark can be destroyed by a variety of distortive attacks. Of the 35 SANDMARK obfuscations, the Graph Coloring-Based watermarking algorithm is theoretically vulnerable to 28. Those obfuscations which will leave the watermark intact include branch inverter, constant pool reorderer, false refactor, field assignment, objectify, overload names, and publicizer.

Due to the inconsistent recognition procedure, we were unable to test the strength against the various obfuscators to identify the potential performance implications associated with destroying the watermark. However, there are several obfuscators including dynamic inliner, inliner, and variable reassigner which have
Figure 7.8: Simulation of a collusive attack on the Graph Coloring-Based algorithm using the SANDMARK Diff Tool.

the potential to improve the performance of the application after destroying the watermark.

For the same reasons that the Graph Coloring-Based algorithm is susceptible to a variety of distortive attacks, it is also susceptible to additive attacks. If an application is double watermarked, it will completely erase the original mark. This is because a new register allocation is applied to the program. Additionally, any watermarking technique which introduces new code that alters the inference graph will destroy the original watermark.

Even though the Graph Coloring-Based algorithm demonstrates a high degree of stealth, the SANDMARK Diff Tool can be leveraged to launch a collusive attack. To simulate such an attack we watermarked the CaffeineMark benchmark application with two different 16-bit watermarks. Using the literal diff algorithm three methods were found to have less than 100% similarity, but the instruction sequences are identical. Figure 7.8 shows that the only difference between the two versions is the local variable assignment.
7.4 Spread Spectrum Watermarking

Spread Spectrum software watermarking maintains the closest link to the techniques used in media watermarking. In fact, the technique is a direct extension of classical spread spectrum watermarking to the code domain. In the classical technique the watermark is encoded as a narrow band signal which is incorporated into the much larger bandwidth signal of the media. Because the watermark introduces only imperceptible modifications to the signal, detection of its presence is difficult.

Stern et al. [93] extended the classic spread spectrum technique to watermarking software by modifying the statistical properties of the application. A vector of instruction-group frequencies is extracted from the program. The instruction groups are replaced with semantically equivalent instructions which have different statistical properties. This algorithm has also been explored by Sahoo and Collberg [87] and Hachez [49].

7.4.1 Embedding

The first phase of the watermark embedding is to extract a vector $c$ representing the frequency $c_i$ for each ordered group of matching language instructions. This is accomplished using the VectorExtraction algorithm:

**Algorithm VectorExtraction**(program):

1. Vector $S \leftarrow n$ ordered groups of machine language instructions;
2. Vector $c$;
3. for each $s_i \in S$ do
4. \hspace{1em} \(c[i] \leftarrow \text{frequency of } s_i \in \text{program};\)
5. return $c$;

The watermark vector $w = (w_1, \ldots, w_n)$, whose coefficients are randomly distributed following a normal law with standard deviation $\alpha$, is embedded by iteratively modifying the statistical properties of the code such that when the vector $\overline{c}$ is extracted $\overline{c} = c + w$. 
Algorithm Embed(program, watermark):
1. Vector $c \leftarrow \text{VectorExtraction}(\text{program})$;
2. Vector $w \leftarrow \text{computeWMVector}(\text{watermark})$;
3. $\overline{c} = c + w$;
4. repeat
   5. modify program;
   6. $c \leftarrow \text{recomputeFrequencies}(\text{program})$;
   7. while $c$ is still making progress towards $\overline{c}$

Modifying the statistical properties during embedding requires the aid of a code book. The code book is manually constructed based on profiling a set of benchmark applications to obtain a table of frequently occurring code patterns. The code book consists of instruction segments which can be used to modify the instruction frequencies. For example, the code book consists of two categories of code patterns:

- **Insertion** patterns consist of two separate code segments, Embed and Nullify. Embed increases the frequency of a vector group by introducing a semantic change. Nullify is then placed on every execution path from Embed to account for the semantic change.

- **Substitution** patterns increase vector group frequencies by providing an alternate, semantically equivalent instruction segment, to replace the selected code segment.

Figure 7.9 illustrates possible code book entries.

### 7.4.2 Recognition

Watermark recognition is accomplished using an informed recognizer with the following signature:

$$\text{recognize}(P', P, k, w, fuzz) \rightarrow \mathbb{B}.$$  

To determine if the watermark is contained in the application we extract the vectors $c$ and $d$ from the original and the watermarked applications respectively. The sim-
Figure 7.9: Possible Spread Spectrum code book entries for both *insertion* and *substitution* code patterns. This figure adapted from Sahoo and Collberg [87].

\[
sim = \sum_{i=1}^{n} \frac{(d_i - c_i)w_i}{\sqrt{(d_i - c_i)^2}}
\]

The key aspect of the recognition procedure is the method for computing similarity between the vectors. The similarity measure used can have an impact on the overall strength of the watermarking technique. Stern et al. suggest the similarity measure

\[
sim = \sum_{i=1}^{n} \frac{(d_i - c_i)w_i}{\sqrt{(d_i - c_i)^2}}
\]
It is suggested by Sahoo and Collberg, that this similarity measure fails to relate recognition with testing for the presence of a transmitted signal within a received signal. To correct this failure they suggest a similarity measure which calculates the normalized linear correlation between two vectors.

\[ \tilde{e} = \frac{e}{|e|} \]

\[ \tilde{w} = \frac{w}{|w|} \]

\[ \text{sim} = \frac{1}{n} \sum_{i=1}^{n} \tilde{e}_i \tilde{w}_i \]

where \( e_i \) and \( w_i \) denote the \( i \)th component in the respective vectors and \( n \) is the number of components in each vector. Using this measure they make the assumption that any correlation ratio greater than 0.9 indicates watermark detected, 0.6-0.9 inconclusive, and less than 0.6 no watermark present.

### 7.4.3 Evaluation

An evaluation of the Spread Spectrum watermarking technique has been performed by Sahoo and Collberg [87] using the SANDMARK tool. In this section we draw on their evaluation and perform additional experiments so that the results can be compared with the other watermarking algorithms presented in this chapter.

**Credibility**  The Spread Spectrum watermarking algorithm uses an informed recognizer which computes the probability that the given watermark is contained in the application. With this type of recognition procedure there is the possibility of both false detections and missed detections. In the SANDMARK implementation the probability of a false detection is dependent on the length of the watermark vector and the values of the components in the watermark vector. Using the specJVM benchmark application, Sahoo and Collberg were able to induce the algorithm into producing false detections (i.e. correlation threshold greater than 0.9) by supplying
the recognizer with random watermark values.

**Data-rate** The data-rate of the **Spread Spectrum** watermarking algorithm is tightly linked to the number of instruction groups defined in the code book. Each instruction group encodes a number of bits. As the size of an application increases the possibility of embedding additional bits increases. However, a larger application does not guarantee a higher data-rate due to the constraints imposed on inserting and substituting code segments. Despite the constraints, using the ten test applications and the **SandMark** implementation, we were able to embed watermarks of varying sizes up to the maximum tried of 64 bytes. The capability to embed a 512 bit watermark in even small applications is a very favorable data-rate. The high data-rate for such small applications was accomplished through the use of method overloading. To increase the frequency of a particular vector group a new over- loaded method is added to the application. This technique not only increases the selected vector group but may create more opportunities to use the insertion and substitution techniques.

**Stealth** Sahoo and Collberg perform an extensive stealth evaluation for the **Spread Spectrum** watermarking algorithm. In their evaluation they looked at two different forms of statistical stealth: global and local. Global stealth measures the difference in the statistical properties of the watermarked program versus that of typical programs. Local stealth measures the difference in the statistical properties of the watermarked methods versus the unaltered methods within that same program. In computing both global and local stealth, Halstead’s [50], McCabe’s [66], and Munson’s [72] method level complexity metrics were used.

In almost all cases, a decrease in both global and local stealth was detected after the application was watermarked. Because of the changes made to the application during the embedding process, it is to be expected that the complexity of the application would increase, thus leading to a decrease in stealth. It is somewhat unclear how this quantitative measure of stealth actually relates to the overall stealth of the
Spread Spectrum watermarking algorithm. For this measure to be useful we need to know what percentage decrease in local or global stealth actually aids an attacker.

**Overhead** As expected, the size impact incurred due to the SANDMARK implementation of the Spread Spectrum algorithm is more significant for the smaller applications like probe and lu. For these applications the average size increase was 21.9% and 18.9% respectively. The overall average size increase is a modest 10%. Figure 7.10 illustrates the size increase measured for the ten test applications using watermark values of size 32-, 64-, 128-, 256-, and 512-bits. From this figure we can also see that the size of the watermark is not directly related to the increase in program size, ie. an increase in watermark size does not necessarily lead to an increase in program size.

For both Java Grande and CaffeineMark a very minimal performance slowdown was recorded. Embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the Java Grande application led to a slowdown of 1.1%, 1.4%, 0.9%, 1.0%, and 0% respectively. Similar, but slightly slower results were obtained for CaffeineMark. Table 7.5 shows the results from embedding 32-, 64-, 128-, 256-, and 512-bit watermarks. While the overall slowdown is minimal, less than 6%, the individual program Loop was more significantly impacted, with a slowdown of greater than 11%.
<table>
<thead>
<tr>
<th>Category</th>
<th>32-bit</th>
<th>64-bit</th>
<th>128-bit</th>
<th>256-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sieve</td>
<td>-2.5%</td>
<td>0.2%</td>
<td>1.8%</td>
<td>1.5%</td>
<td>1.7%</td>
</tr>
<tr>
<td>Loop</td>
<td>15.8%</td>
<td>18.2%</td>
<td>13.9%</td>
<td>11.5%</td>
<td>11.6%</td>
</tr>
<tr>
<td>Logic</td>
<td>0%</td>
<td>0%</td>
<td>-0.3%</td>
<td>0%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>String</td>
<td>0.2%</td>
<td>0.3%</td>
<td>1.8%</td>
<td>-1.5%</td>
<td>-4.8%</td>
</tr>
<tr>
<td>Float</td>
<td>-0.2%</td>
<td>1.3%</td>
<td>-1.7%</td>
<td>0.3%</td>
<td>-0.5%</td>
</tr>
<tr>
<td>Method</td>
<td>10.0%</td>
<td>-0.5%</td>
<td>11.3%</td>
<td>0%</td>
<td>0.1%</td>
</tr>
<tr>
<td>Overall</td>
<td>4.1%</td>
<td>3.5%</td>
<td>5.8%</td>
<td>2.1%</td>
<td>1.4%</td>
</tr>
</tbody>
</table>

Table 7.5: CaffeineMark scores before and after embedding 32-, 64-, 128-, 256-, and 512-bit watermarks using the Spread Spectrum algorithm.

**Robustness**  As with the previous watermarking algorithms presented, the Spread Spectrum watermarking algorithm is vulnerable to distortive attacks. Any attack which alters the watermark vector group frequencies will damage the watermark. If we examine the effects that each of the 35 SandMark obfuscations could possibly have on an application, a watermark embedded using the Spread Spectrum watermarking algorithm could be damaged by 22 of them. These include array folder, block marker, boolean splitter, branch inverter buggy code, class splitter, duplicate registers, dynamic inliner, exception branches, field assignment, inliner, insert opaque predicates, interleave methods, irreducibility, merge local integers, method merger, objectify, opaque branch insertion, promote primitive registers, promote primitive types, random dead code, and reorder instructions. Of course, the actual effects of the obfuscation depend on the application and the code book.

To evaluate the practical effects of the obfuscations we watermarked nine test applications. Of the 35 SandMark obfuscations 19 were able to destroy the embedded watermark in at least one application. Results from applying the obfuscations to the nine test applications are shown in Table 7.6.

For each obfuscation which destroyed the watermark embedded in Java Grande we evaluated the effects the obfuscation had on performance. For this particular testcase 11 obfuscations damaged the watermark to a point where is was not recov-
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array folder</td>
<td>⊕</td>
</tr>
<tr>
<td>Array splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Block marker</td>
<td>⊕</td>
</tr>
<tr>
<td>Bludgeon signatures</td>
<td>⊕</td>
</tr>
<tr>
<td>Boolean splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Branch inverter</td>
<td>⊕</td>
</tr>
<tr>
<td>Buggy code</td>
<td>⊕</td>
</tr>
<tr>
<td>Class splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Constant pool reorderer</td>
<td>⊕</td>
</tr>
<tr>
<td>Duplicate registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Dynamic inliner</td>
<td>⊕</td>
</tr>
<tr>
<td>Exception branches</td>
<td>⊕</td>
</tr>
<tr>
<td>False refactor</td>
<td>⊕</td>
</tr>
<tr>
<td>Field assignment</td>
<td>⊕</td>
</tr>
<tr>
<td>Inliner</td>
<td>⊕</td>
</tr>
<tr>
<td>Insert opaque predicates</td>
<td>⊕</td>
</tr>
<tr>
<td>Interleave methods</td>
<td>⊕</td>
</tr>
<tr>
<td>Irreducibility</td>
<td>⊕</td>
</tr>
<tr>
<td>Merge local integers</td>
<td>⊕</td>
</tr>
<tr>
<td>Method merger</td>
<td>⊕</td>
</tr>
<tr>
<td>Objectify</td>
<td>⊕</td>
</tr>
<tr>
<td>Opaque branch insertion</td>
<td>⊕</td>
</tr>
<tr>
<td>Overload names</td>
<td>⊕</td>
</tr>
<tr>
<td>Parameter aliases</td>
<td>⊕</td>
</tr>
<tr>
<td>Promote primitive registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Promote primitive types</td>
<td>⊕</td>
</tr>
<tr>
<td>Publicizer</td>
<td>⊕</td>
</tr>
<tr>
<td>Random dead code</td>
<td>⊕</td>
</tr>
<tr>
<td>Rename registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>⊕</td>
</tr>
<tr>
<td>Reorder parameters</td>
<td>⊕</td>
</tr>
<tr>
<td>Split classes</td>
<td>⊕</td>
</tr>
<tr>
<td>Static method bodies</td>
<td>⊕</td>
</tr>
<tr>
<td>String encoder</td>
<td>⊕</td>
</tr>
<tr>
<td>Variable reassigner</td>
<td>⊕</td>
</tr>
</tbody>
</table>

⊕ : watermark found in all test applications
⊕ : watermark destroyed in at least one test application

Table 7.6: Results of applying the SANDMARK obfuscations on nine of the test applications which were watermarked using the Spread Spectrum algorithm.
erable. These include array folder, buggy code, dublicate registers, insert opaque predicates, irreducibility, merge local integers, opaque branch insertion, promote primitive registers, promote primitive types, and random dead code. The effect on performance ranged from -1\% (ie. a 1\% speed up) for irreducibility to 281.1\% for promote primitive types to 4104.7\% for exception branches.

Since the Spread Spectrum technique relies on the statistical properties of a program, it is also vulnerable to additive attacks. The level of vulnerability is tightly linked to the contents of the codebook. For example, if one of the codebook entries is used to increase the frequency of the goto instruction, then the DM algorithm can be used to successfully attack the watermarked application. Theoretically, the Spread Spectrum algorithm is susceptible to an additive attack in which any one of the 11 watermarking algorithms, presented in this chapter, is used. However, careful codebook construction could easily prevent the DM and Graph Coloring-Based algorithms from destroying the watermark. Unfortunately, the Spread Spectrum algorithm cannot prevent an additive attack in which the goal of the adversary is to cast doubt on the ownership by introducing additional watermarks.

It is also possible to launch a successful collusive attack against applications which have been watermarked using the Spread Spectrum technique. In addition to locating the watermark, the collusive attack can reveal patterns contained in the code. Such information makes it possible for an attacker to construct a class attack. To illustrate a collusive attack in which the adversary could gain knowledge of the codebook entries, we watermarked the decode application with two different watermark values. We then used the Baker-Manber comparison algorithm within the SandMark Diff Tool. Figure 7.11 shows the bytecode instructions for the method mult in both versions of the application. The differences in the methods are highlighted. Based on just this method the attacker now knows that the instruction sequence new, dup is an entry in the codebook. Additional methods will provide the attacker with even more information about the codebook entries.
7.5 Dummy Method

The Dummy Method algorithm was proposed by Monden et al. [68, 69] for watermarking Java applications. The algorithm embeds a watermark by manipulating the instructions of a dummy method which is added to the application. The insertion of a dummy method creates a degree of freedom in the embedding process since it will never be executed. The instructions in the method can be modified in any way as long as syntactic correctness and type consistency are maintained.

7.5.1 Embedding

The embedding process is accomplished through a series of three phases: dummy method injection, compilation, and watermark injection. For the dummy method injection phase, the algorithm can either require the application programmer to...
create the dummy method at the source code level, or a method can be automatically
generated by the watermarking algorithm at the bytecode level. For the compilation
phase, the application should be compiled using a standard Java compiler which
compiles down to Java bytecode that can be run on the Java Virtual Machine. The
final phase of the embedding process is the actual watermark injection. In this
phase, the binary bit sequence of the watermark is embedded by modifying the
instructions in the dummy method. Monden et al. outline two techniques:

1. Overwrite the numerical operands of certain instructions. For example, the
BIPUSH instruction is followed by a byte value to push onto the stack. Since
the dummy method will never be executed, we can overwrite this byte to
represent one byte (8 bits) of the watermark bit sequence.

2. Replace the opcodes. For example, arithmetic instructions such as IADD,
ISUB, IMUL, etc. can be replaced by other such arithmetic instructions.
There are 8 such integer arithmetic instructions which operate on two
operands. Each instruction can be assigned a unique 3 bit value, thus each
arithmetic instruction in the dummy method represents 3 bits of the water-
mark.

If the dummy method is not large enough to encode the entire watermark, additional
instructions can be appended to the method. In summary, the Embed algorithm is
as follows:
Algorithm Embed(program, watermark):
1. index ← 0;
2. Method dm ← getDummyMethod();
3. for each inst ∈ dm do
   4. index ← alterInst(inst, index, watermark);
5. while index < |watermark| do
6. append new randomly selected instruction;
7. index ← alterInst(inst, index, watermark);

Algorithm alterInst(inst, index, watermark):
1. if inst is a constant value operand instruction then
2. byte ← getByte(index, watermark);
3. replace numerical operand with byte;
4. index += 8;
5. else if inst is an arithmetic instruction then
6. key ← getKey(index, watermark);
7. newInst ← getInst(key);
8. replace inst with newInst;
9. index += 3;
10. return index;

One technique used to hide the superfluous nature of the dummy method is to add a call to the method somewhere in the application. The method call is protected through the use of an opaque predicate [37].

7.5.2 Recognition

To recognize the watermark, the instructions that were used for encoding are identified, converted into their assigned bits, and then concatenated to build the bit sequence representation of the watermark. Once the bit sequence representation is recovered it is decoded.
Algorithm Recognize\( (\text{program}) \):
1. Method \( dm \leftarrow \text{getDummyMethod}(\text{program}) \);
2. for each \( \text{inst} \in dm \) do
3. \hspace{1em} if \( \text{inst} \) is constant value operand \( \text{inst} \) then
4. \hspace{2em} byte \leftarrow \text{getOperandByte}(\text{inst}) \);
5. \hspace{2em} \text{concatByte}(\text{watermark}, \text{byte}) \);
6. \hspace{1em} else if \( \text{inst} \) is arithmetic \( \text{inst} \) then
7. \hspace{2em} bits \leftarrow \text{getInstValue}(\text{inst}) \);
8. \hspace{2em} \text{concatBits}(\text{watermark}, \text{bits}) \);
9. return \( \text{watermark} \);

In practice there is no way to know which method contains the watermark, therefore the recognition procedure is repeated for every method. To identify which recovered string is the watermark the authors make the assumption that if one of the strings is meaningful then it must be the watermark. However, a more credible technique is to prepend a magic character prior to encoding the watermark as a bit sequence. This character aids in eliminating false watermarks.

7.5.3 Evaluation

The first empirical evaluation of the Dummy Method watermarking algorithm was performed by Myles et al. [77]. In this section, we combine results obtained from that evaluation, with a few additional experiments.

Credibility The Dummy Method watermarking algorithm uses a blind extractor to recognize the watermark. This recognition technique is guaranteed to recover the watermark in a watermarked application which is otherwise unaltered. Because the recognition procedure is repeated for every method in the application, there is the possibility of recovering a false watermark. This possibility is considered minimal due to the use of the unusual instruction sequence and the prepended magic character. In our evaluation we were unable to induce the recovery of a false watermark.
value in a watermarked, but otherwise unaltered, application. However, we did recover false watermarks when various obfuscations were applied to unwatermarked applications.

**Data-rate** The data-rate is dependent on the number of instructions in the dummy method which can be altered. However, if the method contains too few instructions, additional random instructions can be added to the method. This enables the embedding of a watermark of any length. Unfortunately, the larger the method becomes the less conspicuous it is.

**Stealth** There are a variety of aspects which compromise the stealth of the dummy methods algorithm. In the SDM implementation the dummy method is automatically injected by copying an existing method. This makes it possible to watermark any Java application, not just ones written by programmers who planned ahead. However, automation also potentially decreases the stealth. A specifically constructed method could decrease the possibility of creating a highly unusual instruction sequence during embedding.

After embedding, the instructions contained in the dummy method can be quite unstealthy. On examination the instructions will often appear to be random because the opcodes have been modified. In some cases, the replaced arithmetic instructions are not even common instructions. For example, Figure 7.12 illustrates a piece of decompiled code from the watermarked fft application. Line (8) shows a \((\text{mod } 1)\) operation as an index to an array.

A final compromise in stealth is the superfluous nature of the dummy method. To improve the stealth a method call can be inserted which is protected by an opaque predicate. This insures that the method is never called and thwarts many static analysis techniques as well as visual inspection. However, an attacker may still discover that the method is never executed through dynamic analysis.
(1) int21 = 0x21 / int46 / int19;
(2) int47 = 2 / (int46 * int19 - int5);
(3) double23 = double_1darray1[int47];
(4) double25 = double_1darray1[int47 & 0x1];
(5) double27 = double7 * double23 - double9 * double25;
(6) double29 = double7 * double25 + double9 * double23;
(7) double_1darray1[int47] = double_1darray1[int21] - double27;
(8) double_1darray1[int47 % 1] =
(9) double_1darray1[int21 - 1] - double29;
(10) double_1darray1[int21] += double27;
(11) double_1darray1[int21 + 1] += double29;

Figure 7.12: Decompiled code from the fft application after watermarking using Dummy Method algorithm.

**Overhead**  A small increase in code size is recorded because a new method, an opaque predicate, and a call to the method are added. The dummy method is in most cases approximately the size of the largest method in the application. In some cases, it is necessary to add even more instructions to the dummy method, but in general very few extra instructions are needed. Figure 7.13 illustrates the size increase measured for the ten test applications using watermark values of size 32-, 64-, 128-, 256-, and 512-bits. From this figure we can see that embedding even a 512-bit watermark in the largest application Java Grande has no effect. For the smaller applications as the watermark value increased in size a more noticeable size impact was recorded. However, the maximum size increase was only 15% and that was for the smallest application probe.

The negligible effect on performance is due to the fact that the dummy method is never executed. The only additional instructions that are executed in the watermarked program are the opaque predicate instructions. Our implementation uses algebraic predicates and the time required to execute the instructions for the opaque predicates is very small. Embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the Java Grande application led to a slowdown of less than 1% for each watermark value. Table 7.7 shows the results from embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the CaffeineMark benchmark application. The overall change in performance is negligible.
Figure 7.13: Size impact incurred using the **Dummy Method** algorithm.

**Robustness** The **Dummy Method** watermarking algorithm is very susceptible to subtractive attacks because of its poor stealth. For most programs we evaluated, the dummy method which held the watermark was the largest method in the class. By carefully inspecting the call(s) to this method, the attacker could verify that the dummy method is never actually executed. In addition, the unusual instruction sequence will draw attention to the method. Once the attacker realizes the method is never executed it can be removed thus eliminating the watermark.

The **Dummy Method** algorithm is also fairly vulnerable to distortive attacks. The recognition process relies on the order of instructions in the dummy method. Even if the attacker did not know which method contained the watermark, performing distortive attacks that rearrange the instructions will erase at least part of the watermark.

Of the 35 obfuscations in **SandMark**, the **Dummy Method** algorithm is theoretically vulnerable to 16 (block marker, boolean splitter, buggy code, duplicate registers, dynamic inliner, exception branches, field assignment, inliner, insert opaque predicates, interleave methods, irreducibility, method merger, opaque branch insertion, promote primitive registers, promote primitive types, and reorder instructions). When the **SandMark** obfuscations are applied to the nine test applications, 13 de-
destroyed the watermark in at least one application. The results from applying the obfuscations are shown in Table 7.8. For the Java Grande benchmark application only one of the obfuscations successfully destroyed the watermark: opaque branch insertion. This obfuscation had only a moderate impact on performance at 18\%. However, as with previous algorithms, the Dummy Method algorithm is susceptible to an attack which uses method inlining. In this case, the watermark can be destroyed while improving the performance of the application.

The Dummy Method algorithm is also vulnerable to additive attacks. If an application is double watermarked using the Dummy Method algorithm, the original watermark will not be damaged. However, the presence of the second watermark will make it more difficult to prove ownership. Additionally, a watermarking algorithm which rearranges the instructions in the dummy method will damage the watermark. Such damage is less likely to occur than with a distortive attack. This is because a distortive attack is generally applied to all methods in the application, where an additive attack may only alter small pieces of the application, thus, the dummy method may not be changed during the attack. Eleven watermarking algorithms are presented in this chapter. Of these algorithms, eight have the potential to destroy the original watermark. These include DM, Moskowitz-Cooperman, Graph Coloring-Based, Spread Spectrum, Graph-Theoretic, Arboit, Collberg-Thomborson, Path-Based,

<table>
<thead>
<tr>
<th>Category</th>
<th>32-bit</th>
<th>64-bit</th>
<th>128-bit</th>
<th>256-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sieve</td>
<td>0.3%</td>
<td>0.4%</td>
<td>0.3%</td>
<td>0.6%</td>
<td>0%</td>
</tr>
<tr>
<td>Loop</td>
<td>0.4%</td>
<td>0.6%</td>
<td>-0.3%</td>
<td>-1.5%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>Logic</td>
<td>1.4%</td>
<td>0.9%</td>
<td>0.4%</td>
<td>1.6%</td>
<td>0.3%</td>
</tr>
<tr>
<td>String</td>
<td>-13.4%</td>
<td>2.4%</td>
<td>0.3%</td>
<td>0%</td>
<td>-1.4%</td>
</tr>
<tr>
<td>Float</td>
<td>1.4%</td>
<td>-1.5%</td>
<td>-5.5%</td>
<td>0.2%</td>
<td>-0.2%</td>
</tr>
<tr>
<td>Method</td>
<td>-0.2%</td>
<td>-0.3%</td>
<td>-0.5%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Overall</td>
<td>-2.3%</td>
<td>0.4%</td>
<td>-1.0%</td>
<td>0.1%</td>
<td>-0.3%</td>
</tr>
</tbody>
</table>

Table 7.7: CaffeineMark scores before and after embedding 32-, 64-, 128-, 256-, and 512-bit watermarks using the Dummy Method algorithm.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array folder</td>
<td>⊕</td>
</tr>
<tr>
<td>Array splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Block marker</td>
<td>⊕</td>
</tr>
<tr>
<td>Bludgeon signatures</td>
<td>⊕</td>
</tr>
<tr>
<td>Boolean splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Branch inverter</td>
<td>⊕</td>
</tr>
<tr>
<td>Buggy code</td>
<td>⊕</td>
</tr>
<tr>
<td>Class splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Constant pool reorderer</td>
<td>⊕</td>
</tr>
<tr>
<td>Duplicate registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Dynamic inliner</td>
<td>⊕</td>
</tr>
<tr>
<td>Exception branches</td>
<td>⊕</td>
</tr>
<tr>
<td>False refactor</td>
<td>⊕</td>
</tr>
<tr>
<td>Field assignment</td>
<td>⊕</td>
</tr>
<tr>
<td>Inliner</td>
<td>⊕</td>
</tr>
<tr>
<td>Insert opaque predicates</td>
<td>⊕</td>
</tr>
<tr>
<td>Interleave methods</td>
<td>⊕</td>
</tr>
<tr>
<td>Irreducibility</td>
<td>⊕</td>
</tr>
<tr>
<td>Merge local integers</td>
<td>⊕</td>
</tr>
<tr>
<td>Method merger</td>
<td>⊕</td>
</tr>
<tr>
<td>Objectify</td>
<td>⊕</td>
</tr>
<tr>
<td>Opaque branch insertion</td>
<td>⊕</td>
</tr>
<tr>
<td>Overload names</td>
<td>⊕</td>
</tr>
<tr>
<td>Parameter aliases</td>
<td>⊕</td>
</tr>
<tr>
<td>Promote primitive registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Promote primitive types</td>
<td>⊕</td>
</tr>
<tr>
<td>Publicizer</td>
<td>⊕</td>
</tr>
<tr>
<td>Random dead code</td>
<td>⊕</td>
</tr>
<tr>
<td>Rename registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>⊕</td>
</tr>
<tr>
<td>Reorder parameters</td>
<td>⊕</td>
</tr>
<tr>
<td>Split classes</td>
<td>⊕</td>
</tr>
<tr>
<td>Static method bodies</td>
<td>⊕</td>
</tr>
<tr>
<td>String encoder</td>
<td>⊕</td>
</tr>
<tr>
<td>Variable reassigner</td>
<td>⊕</td>
</tr>
</tbody>
</table>

⊕ : watermark found in all test applications  
⊗ : watermark destroyed in at least one test application

Table 7.8: Results of applying the SANDMARK obfuscations on 9 of the test applications which were watermarked using the Dummy Method algorithm.
and Abstract Interpretation.

The Dummy Method algorithm is also highly susceptible to collusive attacks. This attack was demonstrated in the previous evaluation [77] using the 1u application. The application was watermarked with two different strings of length 5. Since the original application is the same, the same method was chosen by the algorithm to copy. Using the Diff Tool, the dummy methods were found to be 97.6% similar. The dummy methods were also found to be about 94% similar to the copied method. When comparing the dummy methods, the only differences were in the arithmetic instructions and the operands of certain instructions.

7.6 Graph-Theoretic Watermarking

Computer programs contain many different types of graphs, e.g. interference and control-flow graphs. As can be seen from previously described watermarking techniques, these graphs can be manipulated to incorporate a watermark. Venkatesan, Vazirani, and Sinha [100] describe an alternate method of incorporating a watermark into a program through graph manipulation. The Graph-Theoretic Watermarking (GTW) algorithm embeds the watermark by augmenting a control flow graph. The watermark is itself encoded as a flow-graph which becomes a subgraph in the larger control flow graph.

7.6.1 Embedding

In the first step of the GTW algorithm, the watermark is converted into one or more control flow graphs by generating executable code for each basic block. Venkatesan et al. fail to provide the necessary implementation details for the construction of the watermark graph. Such details were, however, filled in by Collberg et al. [32] in their implementation of GTW within the SANDMARK framework (GTWSM). To construct the watermark graph in the GTWSM algorithm the watermark $w$ is split into a multiset $S$ of $k$ integers, $k \geq 2$. Each $s_i \in S$ is then encoded as a digraph which meets the following requirements:
1. The graph structure must be a valid CFG.
2. The graph should have a maximum out-degree of two.
3. The graph should be reducible.
4. The graph should not be deeply nested.

The particular graph type used in $\text{GTW}_{SM}$ is the reducible permutation graph [31]. For each node in the graph a set of instructions is generated which forms a basic block. The basic blocks are connected using conditional branches and fall-throughs wherever possible.

In the second step the watermark graph is connected to the original CFG. This step poses the greatest vulnerability for the algorithm. The authors conjecture that a weakly connected watermark graph can easily be identified using standard graph algorithms or even possibly through visual inspection. To construct a strong connected watermark graph the original CFG is partitioned into $n$ clusters. Edges are then added between the clusters and the watermark graphs using a random walk heuristic.

\textbf{Algorithm} \texttt{EdgeAdding}($\text{graph}$, $\text{clusters}$, $m$):

\begin{enumerate}
    \item Cluster $c \leftarrow \texttt{chooseRandomCluster} (\text{clusters})$;
    \item Node $n \leftarrow \texttt{chooseRandomNode}(c)$;
    \item $\text{edgesAdded} \leftarrow 0$;
    \item while $\text{edgesAdded} < m$ do
        \begin{enumerate}
            \item $c \leftarrow \texttt{chooseRandomCluster} (\text{clusters} - c)$;
            \item Node $l \leftarrow \texttt{chooseRandomNode}(c)$;
            \item $\text{graph}.\text{addEdge}(n, l)$;
            \item $n \leftarrow l$;
            \item $\text{edgesAdded}++$;
        \end{enumerate}
\end{enumerate}

As with previous watermarking techniques, executing the watermark code could have adverse effects on program behavior. Therefore, the new edges must be prevented from executing through techniques such as opaque predicates.
The final step in the embedding process is to secretly mark the watermark nodes so they can be found during watermark recovery. The authors do not suggest a specific marking technique. Instead they say that one or more bits can be stored in the node to flag it as part of the watermark. Collberg et al. provide specific examples of how block marking can be accomplished. These include, but are not limited to:

1. Compute a checksum of the instructions and use one or more bits of that as a mark.

2. Add code that serves no purpose other than as a marker.

3. Count the number of instructions in the block using the parity as a mark.

4. Transform the instruction sequence in each block to a canonical form.

This step is necessary because the embedding process tries to make the watermark graph(s) indistinguishable from the actual program. Putting the embedding steps together yields the following Embed algorithm:

Algorithm Embed(program, watermark):
1. \(pCFG \leftarrow \text{constructCFG}(\text{program})\);
2. \(\text{int}[] w \leftarrow \text{splitWM}(\text{watermark})\);
3. \(\text{for } i \leftarrow 0 \text{ to } w.\text{length do}
4. \quad g \leftarrow \text{constructGraph}(w_i);
5. \quad \text{wmCFG} \leftarrow \text{generateCFG}(g);
6. \quad pCFG.\text{add}(\text{wmCFG});
7. \quad \text{clusters} \leftarrow \text{computeClusters}(pCFG);
8. \quad \text{AddEdges}(pCFG, \text{clusters}, m);
9. \quad \text{markNodes}(pCFG, \text{wmCFG.\text{nodes}});

Figure 7.14 illustrates the GTW watermarking transformation. In this example, the checked blocks represent the watermark code in which the blocks have been marked to aid in recognition.
7.6.2 Recognition

The recognition process relies on the fact that the watermark blocks were marked during the embedding process. To recover the watermark, the first step is to identify each of the marked blocks. During embedding watermarked clusters of basic blocks were added to the original CFG. To recover the watermark these clusters must be reassembled. Venkatesan et al. call this process subset sampling. The subsets of the marked blocks are split into samples. In the GTW\textsubscript{SM} algorithm the samples are function level control flow graphs. Based on the samples, each graph is decoded to the value it represents. The recovered set of values in then combined to form the watermark.

Algorithm Recognize(program):
1. $pCFG \leftarrow$ constructProgramCFG(program);
2. for each $b_i \in pCFG$ do
3. if $b_i$ is marked then
4. markedBlocks.add($b_i$);
5. samples $\leftarrow$ formSubsetSamples(markedBlocks);
6. for each $g \in$ samples do
7. $v \leftarrow$ decode($g$);
8. values.add($v$);
9. watermark $\leftarrow$ computeWM(values);
10. return watermark;
7.6.3 Evaluation

To formulate an evaluation of the GTW algorithm, which is suitable for comparison purposes, we draw on the results presented by Collberg et al. [32], as well as several additional experiments which we performed.

Credibility The GTW algorithm uses a blind recognizer to extract the watermark value from the program. As with other watermarking algorithms which use this type of recognizer, the GTW algorithm is guaranteed to recover the watermark from a watermarked, but otherwise unaltered, program.

Data-rate To embed a watermark, the GTW algorithm adds additional code to the program. This type of embedding technique enables any size watermark to be embedded in any application. However, a larger watermark requires a larger watermark graph or a greater number of watermark graphs. For small applications such graphs could become suspicious.

Stealth There are a variety of aspects which compromise the stealth of the SANDMARK implementation of the GTW algorithm. To watermark the application, GTW$_{SM}$ adds several artificially-generated methods. These methods have a higher percentage of arithmetic instructions than general methods written in Java. Based on analysis by Collberg et al. [34] the most frequently occurring arithmetic instruction in a Java method is iadd. This instruction occurs with a frequency of 0.6%. In general, arithmetic instructions only account for 1% of the instructions in a method. However, the methods inserted by GTW$_{SM}$ consist of approximately 20% arithmetic instructions. The bytecode in Figure 7.15 is an example of the type of instruction sequence generated, the arithmetic instructions are highlighted. This method contains 26 instructions, of these six or 23% are arithmetic instructions.

The inserted watermark graphs are constructed such that they are reducible permutation graphs (RPGs). This graph structure was chosen because it is designed to resemble the structure of actual control flow graphs. Despite this fact, Collberg
Figure 7.15: Example instruction sequence generated by the $\text{GTW}_{SM}$ algorithm.

et al. [32] discovered in their evaluation of $\text{GTW}_{SM}$ that only 2 of 3236 methods in the SpecJVM benchmarking suite contain CFGs which are RPGs.

**Overhead** Of the watermarking algorithms presented thus far, the $\text{SANDMARK}$ implementation of GTW has had the most significant impact on both application size and performance. Figure 7.16 illustrates that the average increase in size is approximately 100%. As can be expected, the largest application, $\text{Java Grande}$ incurred the lowest size increase of approximately 17% for the various size watermarks.

Unlike the Dummy Method algorithm, the code added using GTW is actually executed. Because of this a significant slowdown was observed for both $\text{Java Grande}$ and $\text{CaffeineMark}$. Embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the $\text{Java Grande}$ application led to a slowdown of 73.1%, 108.2%, 68.6%, 133.3%, and 61.1% respectively. Table 7.9 shows the results of embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the $\text{CaffeineMark}$ application where the overall slowdown was at least 48%.
Robustness  Those features which compromise the stealth of the algorithm are the same features which can be leveraged by an adversary in a subtractive attack. Removing the watermark is as simple as identifying and removing the artificially generated methods. Since these methods currently contain a much higher percentage of arithmetic instructions, an attacker can easily identify them through static analysis.

The GTW$_{SM}$ algorithm is also moderately susceptible to distortive attacks. Any modification which makes changes to a control flow graph is capable of damaging the watermark. If we consider the 35 SANDMARK obfuscations, the GTW$_{SM}$ algorithm is theoretically vulnerable to 14 of them. These obfuscations include block marker, bludgeon signatures, buggy code, interleave methods, irreducibility, merge local integers, method merger, objectify, opaque branch insertion, promote primitive registers, promote primitive types, random dead code, splitting classes, and exception branches.

In practice we found that six of the SANDMARK obfuscations were able to destroy the embedded watermark in at least one of the nine test applications. Results from applying the obfuscations are shown in Table 7.10. The table shows that buggy code, interleave methods, irreducibility, opaque branch insertion, random dead code, and
Table 7.9: CaffeineMark scores before and after embedding 32-, 64-, 128-, 256-, and 512-bit watermarks using the GTW$_{SM}$ algorithm.

---

exception branches destroyed the watermark. When the obfuscations were applied to the watermark Java Grande application all six successfully damanged the watermark to a point that it was not recoverable. The performance implications for destroying the watermark in this application ranged from 2% slowdown for random dead code to 1816% when exception branches are blindly inserted.

Despite the moderate resistance to distortive attacks, an adversary could essentially replace the original proof of ownership with his own through an additive attack. Of the 11 watermarking algorithms presented in this chapter, nine have the potential to damage the GTW$_{SM}$ watermark. Using the 10 test applications we found that the embedded watermark is still recoverable after additive attacks using DM, Spread Spectrum, Graph Coloring-Based, and Dummy Methods. Even though the original watermark was recovered with these algorithms so was the second watermark, thus casing doubt on the original ownership.

### 7.7 Arboit Algorithm

Arboit [18] proposed two watermarking techniques aimed at Java applications. Both techniques are based on the use of opaque predicates to encode the watermark. The first algorithm (henceforth A1) is the basic insertion algorithm which directly uses the opaque predicates. The second Arboit algorithm (henceforth A2) is similar to
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array folder</td>
<td>⊕</td>
</tr>
<tr>
<td>Array splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Block marker</td>
<td>⊕</td>
</tr>
<tr>
<td>Bludgeon signatures</td>
<td>⊕</td>
</tr>
<tr>
<td>Boolean splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Branch inverter</td>
<td>⊕</td>
</tr>
<tr>
<td>Buggy code</td>
<td>⊕</td>
</tr>
<tr>
<td>Class splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Constant pool reorderer</td>
<td>⊕</td>
</tr>
<tr>
<td>Duplicate registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Dynamic inliner</td>
<td>⊕</td>
</tr>
<tr>
<td>Exception branches</td>
<td>⊕</td>
</tr>
<tr>
<td>False refactor</td>
<td>⊕</td>
</tr>
<tr>
<td>Field assignment</td>
<td>⊕</td>
</tr>
<tr>
<td>Inliner</td>
<td>⊕</td>
</tr>
<tr>
<td>Insert opaque predicates</td>
<td>⊕</td>
</tr>
<tr>
<td>Interleave methods</td>
<td>⊕</td>
</tr>
<tr>
<td>Irreducibility</td>
<td>⊕</td>
</tr>
<tr>
<td>Merge local integers</td>
<td>⊕</td>
</tr>
<tr>
<td>Method merger</td>
<td>⊕</td>
</tr>
<tr>
<td>Objectify</td>
<td>⊕</td>
</tr>
<tr>
<td>Opaque branch insertion</td>
<td>⊕</td>
</tr>
<tr>
<td>Overload names</td>
<td>⊕</td>
</tr>
<tr>
<td>Parameter aliases</td>
<td>⊕</td>
</tr>
<tr>
<td>Promote primitive registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Promote primitive types</td>
<td>⊕</td>
</tr>
<tr>
<td>Publicizer</td>
<td>⊕</td>
</tr>
<tr>
<td>Random dead code</td>
<td>⊕</td>
</tr>
<tr>
<td>Rename registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>⊕</td>
</tr>
<tr>
<td>Reorder parameters</td>
<td>⊕</td>
</tr>
<tr>
<td>Split classes</td>
<td>⊕</td>
</tr>
<tr>
<td>Static method bodies</td>
<td>⊕</td>
</tr>
<tr>
<td>String encoder</td>
<td>⊕</td>
</tr>
<tr>
<td>Variable reassigner</td>
<td>⊕</td>
</tr>
</tbody>
</table>

⊕ : watermark found in all test applications
⊕ ⊕ : watermark destroyed in at least one test application

Table 7.10: Results of applying the SANDMARK obfuscations on nine of the test applications which were watermarked using the GTW$_{SM}$ algorithm.
∀x, y ∈ Z \quad 7y^2 - 1 \neq x^2
∀x ∈ Z \quad 2\lfloor \frac{x^2}{2} \rfloor
∀x ∈ Z \quad 2|x(x + 1)|
∀x ∈ Z \quad x^2 \geq 0
∀x ∈ Z \quad 3|x(x + 1)(x + 2)|
∀x ∈ Z \quad 7 \nexists x^2 + 1
∀x ∈ Z \quad 81 \nexists x^2 + x + 7
∀x ∈ Z \quad 19 \nexists 4x^2 + 4
∀x ∈ Z \quad 4|x^2(x + 1)(x + 1)|

Table 7.11: Sample opaque predicate library.

A1 except opaque methods are used to embed the watermark.

In this section we introduce the original watermarking algorithm as it was presented by Arboit. Through implementing and analyzing the algorithm we developed a thorough understanding of its strengths and weaknesses. Based on this understanding we made extensions to the algorithm which were presented in a previous publication [75] and will be discussed in Chapter 8.

7.7.1 Embedding

To embed a watermark using either A1 or A2 requires a library of opaque predicates. To increase the strength of the algorithm it is necessary that this library remain secret and be of a substantial size. A publicly known library provides an adversary with information which will enable identification of the watermark.

The watermark bits are encoded in the opaque predicates either in their constants or their rank within the library. For example, suppose the library consists of the opaque predicates in Table 7.11. Using constant encoding the opaque predicate 7 \nexists x^2 + 1 will encode the value 7 + 1 = 8. However, using rank it encodes the value 5. To embed a watermark, w is split into k pieces \( w_0, ..., w_{k-1} \). The watermark is split into pieces such that each piece encodes bits of the watermark and its index in the sequence. This enables the pieces to be recovered in any order.

For both algorithms the embedding is accomplished by randomly selecting k branching points, \( b_0, ..., b_{k-1} \), throughout the application. Using algorithm A1, at
each branching point $b_i$, either $\land P^T_{b_i}$, $\lor \neg P^T_{b_i}$, or $\lor P^F_{b_i}$ is appended to the predicate.

**Algorithm Embed**(program, watermark):
1. $\text{int}[k] \ w \leftarrow \text{splitWM}(\text{watermark}, k)$;
2. $\text{branchPoints} \leftarrow \text{chooseBranchPoints}(\text{program}, k)$;
3. for each $b_i \in \text{branchPoints}$ do
   4. $\text{pred} \leftarrow \text{encodeWM}(w_i)$;
   5. append $\text{pred}$ to $b_i$;

To illustrate the A1 Embed algorithm, suppose our watermark is encoded in the opaque predicate $x^2 \geq 0$. A watermark could be embedded as follows:

```java
class C{
    void m1(int a, int b){
        ...
        if(a <= b){...
        else {...
        ...
    }
}
}
```

Using algorithm A2, for each $b_i$, an opaque method $M^T_{b_i}$ or $M^F_{b_i}$ is created and a method call is appended. The bits of the watermark are encoded in the opaque method through the opaque predicate that it evaluates.

**Algorithm Embed**(program, watermark):
1. $\text{int}[k] \ w \leftarrow \text{splitWM}(\text{watermark}, k)$;
2. $\text{branchPoints} \leftarrow \text{chooseBranchPoints}(\text{program}, k)$;
3. for each $b_i \in \text{branchPoints}$ do
   4. $m \leftarrow \text{constructOMethod}(w_i)$;
   5. program.add($m$);
   6. append call to $m$ to $b_i$;

To illustrate, suppose we use the same opaque predicate as above. Using A2 the application would be transformed in the following way:
7.7.2 Recognition

The recognition procedure varies slightly depending on which embedding technique is used. Watermark recovery using A1 involves an exhaustive search of each method. To identify sets of instructions that may be opaque predicates the basic blocks of the CFG and expression trees are constructed. Each opaque predicate will end with an if instruction which can be found as the last instruction of a basic block. The instructions that comprise the expression tree for that if instruction are compared to the entries in the opaque predicate library.

Algorithm Recognize(program):
1. for each \( m_i \in \text{program} \) do
2. \( \text{cfg} \leftarrow \text{constructCFG}(m_i) \);
3. for each block \( \in \text{cfg} \) do
4. if last \( \text{inst} \in \text{block} \) is if instruction then
5. \( \text{exprTree} \leftarrow \text{getExprTree}(\text{inst}) \);
6. if \( \text{exprTree} \) instruction set is in opaque predicate library then
7. \( w_i \leftarrow \text{decode}(\text{exprTree}) \);
8. combine all \( w_i \) to produce \text{watermark};
9. return \text{watermark};
If the watermark was embedded using A2 then each method is scanned looking for instructions which call a method that has the same signature as one of the opaque methods. Within each opaque method is an opaque predicate that is identified using the same technique as in A1.

If \( w_i \) is encoded using rank, the rank of that particular opaque predicate is identified. If constants are used, the sum of the constants is extracted from the predicate. Once all possible \( w_i \) have been identified the values are combined to produce the watermark value.

### 7.7.3 Evaluation

The A1 and A2 algorithms were previously evaluated by Myles and Collberg [75]. In this section we draw on the previous evaluation and we perform additional experiments so as to provide a unified evaluation.

**Credibility** As with previous watermarking algorithms which use an extraction based recognition procedure, both A1 and A2 are guaranteed to recognize a watermark in a watermarked but otherwise unaltered program. Because the algorithms rely on the use of opaque predicates there is the possibility of a false positive. This could occur when an application is obfuscated using a technique which inserts opaque predicates. For a false positive to occur in this situation the inserted opaque predicates must be contained in the opaque predicate library used by the watermarker. To evaluate the algorithms with respect to this property we ran the recognition algorithm on non-watermarked, obfuscated versions of the test applications. No false positives were detected.

**Data-rate** The data-rate for A1 and A2 are roughly the same. This is because the embedding process is based on identifying usable conditional branch instructions. The only embedding detail which can alter the data-rate is whether the watermark is encoded using constants or rank. When rank is used the watermark may need to be split into more pieces since the value of each piece is restricted to the values
0 through $n$, where $n$ is the size of opaque predicate library. In the SandMark implementation $n = 8$. Table 7.12 shows that on average, by using constants roughly 7 times as many characters can be embedded. The table also shows the total number of if instructions found in the applications.

**Stealth** The stealth of both algorithms A1 and A2 is dictated by the opaque predicate library. It is imperative that the library used by the watermarking algorithm remain secret. Additionally, a library of commonly known opaque predicates can be leveraged by an attacker to identify the pieces of the watermark. One technique that can be used to evaluate the stealth is to examine how the static statistics of the application change between non-watermarked and watermarked versions of the application. While this technique does not provide a quantitative measure, it does highlight areas of the watermarked application which might be suspicious to an attacker. Table 7.13 contains some static statistics of watermarked and non-watermarked versions of a tic-tac-toe game (TTT) and the Java Grande benchmark application. What we can see from these statistics is that applications watermarked using A1 more closely resembles the original.

<table>
<thead>
<tr>
<th>Application</th>
<th>Max characters using Constants</th>
<th>Max characters using Rank</th>
<th>Total if instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>27</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>fft</td>
<td>14</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>illness</td>
<td>61</td>
<td>5</td>
<td>104</td>
</tr>
<tr>
<td>lu</td>
<td>15</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>machineSim</td>
<td>64+</td>
<td>5</td>
<td>162</td>
</tr>
<tr>
<td>matrix</td>
<td>27</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>probe</td>
<td>15</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>puzzle</td>
<td>64+</td>
<td>5</td>
<td>154</td>
</tr>
<tr>
<td>Java Grande</td>
<td>64+</td>
<td>13</td>
<td>325</td>
</tr>
<tr>
<td>CaffeineMark</td>
<td>64+</td>
<td>5</td>
<td>113</td>
</tr>
</tbody>
</table>

Table 7.12: Maximum characters embedded when encoding the watermark using constants and rank.
Table 7.13: Static statistics of watermarked and non-watermarked version of TTT and Java Grande. The watermark value is “wildcat”.

**Overhead**  The size impact incurred by either A1 or A2 depends on the number of watermark pieces and therefore on the size of the watermark. In addition, the encoding technique also has an impact on the overhead. When using constants the value of each $w_i$ can be larger which means the watermark does not need to be split into as many pieces. Using A1 roughly 80 bytes are added to the application for each $w_i$. This value increases to 104 bytes when using A2. Figure 7.17 illustrates the size overhead associated with each of the Arboit variations. To produce this comparison we embedded a 32-bit watermark in each application which would hold such a watermark using rank encoding. There were five such test applications. The figure shows that rank encoding has a more significant impact than constant encoding and that A2 has a more significant impact than A1.

Figure 7.18 illustrates the size increase measured for the ten test applications using A1 and A2 with constant encoding. As can be seen there is a direct correlation between increase in watermark size and the increase in application size.

The impact on performance varies with the size of the watermark. Embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the Java Grande application led to a slowdown of 0.9%, 0.8%, 2.5%, 1.8%, and 3.4% for A1 and 0.6%, 0.2%, 1.0%, 2.4%, and 5.2% for A2. Table 7.14 shows the results from embedding 32-, 64-, 128-, 256-, and 512-bit watermarks in the CaffeineMark benchmark application using constant encoding and the A1 and A2 algorithms. For A1 a minimal performance improvement was observed for the 32-bit watermark. However, both algorithms had
Figure 7.17: Size impact incurred using the different variations of the Arboit algorithm and a 32 bit watermark.

a significant slowdown when a 512-bit watermark was embedded.

Robustness One of the first things an adversary may do in an attempt to eliminate a watermark is decompile the application. Once the code has been decompiled the attacker can search for aspects of the code that look suspicious. If the attacker is familiar with simple number theory properties he may realize that the watermark application contains opaque predicates. If the opaque predicates are removed the application will still function normally and the attacker has subverted the protection. The Arboit techniques will always be susceptible to subtractive attacks. However, by using stronger opaque predicates, in particular ones that are not commonly known, detection and removal will be more difficult. In addition, maintaining the secrecy of the opaque predicate library will also improve resiliency against subtractive attacks.

An important assumption made in the study of software watermarking is that the attacker knows the algorithm used to embed the watermark. Based on this assumption both A1 and A2 can easily be attacked by simply applying a transformation that inserts additional opaque predicates throughout the application. In addition to inserting opaque predicates, the algorithms are theoretically susceptible to 17 other SandMark obfuscations. These include block marker, boolean splitter,
Figure 7.18: Size impact using A1 and A2 with constant encoding.

branch inverter, buggy code, constant pool reorderer, dublicate registers, dynamic inliner, exception branches, inliner, interleave methods, irreducibility, merge local integers, method merger, opaque branch insertion, promote primitive registers, promote primitive types, and random dead code.

To evaluate the practical effects of the obfuscations we watermarked the nine test applications using A1 and A2 with constant encoding. Of the 35 SandMark obfuscations 9 were able to destroy a watermark embedded using A1 and 10 destroyed the A2 watermark in at least one of the nine applications. The results from applying the obfuscations are shown in Table 7.15. In the case of the Java Grande
<table>
<thead>
<tr>
<th>Category</th>
<th>32-bit</th>
<th>64-bit</th>
<th>128-bit</th>
<th>256-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sieve</td>
<td>-2.3%</td>
<td>52.0%</td>
<td>52.3%</td>
<td>52.2%</td>
<td>80.4%</td>
</tr>
<tr>
<td>Loop</td>
<td>2.0%</td>
<td>-0.1%</td>
<td>-9.3%</td>
<td>47.5%</td>
<td>50.4%</td>
</tr>
<tr>
<td>Logic</td>
<td>-0.4%</td>
<td>-0.5%</td>
<td>-2.6%</td>
<td>-0.2%</td>
<td>89.2%</td>
</tr>
<tr>
<td>String</td>
<td>-1.1%</td>
<td>-5.1%</td>
<td>-1.5%</td>
<td>0.9%</td>
<td>-4.7%</td>
</tr>
<tr>
<td>Float</td>
<td>-1.1%</td>
<td>5.4%</td>
<td>4.9%</td>
<td>21.4%</td>
<td>27.1%</td>
</tr>
<tr>
<td>Method</td>
<td>0.5%</td>
<td>0.6%</td>
<td>0.5%</td>
<td>8.9%</td>
<td>11.1%</td>
</tr>
<tr>
<td>Overall</td>
<td>-0.4%</td>
<td>11.5%</td>
<td>10.3%</td>
<td>25.0%</td>
<td>55.3%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>32-bit</th>
<th>64-bit</th>
<th>128-bit</th>
<th>256-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sieve</td>
<td>51.9%</td>
<td>51.9%</td>
<td>71.5%</td>
<td>76.7%</td>
<td>71.3%</td>
</tr>
<tr>
<td>Loop</td>
<td>0.1%</td>
<td>-1.2%</td>
<td>6.4%</td>
<td>50.3%</td>
<td>50.5%</td>
</tr>
<tr>
<td>Logic</td>
<td>-1.3%</td>
<td>-1.2%</td>
<td>-0.1%</td>
<td>21.5%</td>
<td>90.0%</td>
</tr>
<tr>
<td>String</td>
<td>-0.2%</td>
<td>-1.9%</td>
<td>-0.4%</td>
<td>-0.9%</td>
<td>-13.6%</td>
</tr>
<tr>
<td>Float</td>
<td>-4.3%</td>
<td>-1.0%</td>
<td>4.8%</td>
<td>28.0%</td>
<td>36.4%</td>
</tr>
<tr>
<td>Method</td>
<td>0.6%</td>
<td>15.0%</td>
<td>14.9%</td>
<td>15.3%</td>
<td>17.9%</td>
</tr>
<tr>
<td>Overall</td>
<td>10.7%</td>
<td>13.1%</td>
<td>22.5%</td>
<td>38.2%</td>
<td>54.5%</td>
</tr>
</tbody>
</table>

Table 7.14: CaffeineMark scores before and after embedding 32-, 64-, 128-, 256-, and 512-bit watermarks using the A1 and A2 algorithms.

application six obfuscations destroyed the A1 watermark and eight destroyed the A2. The performance implications associated with destroying the A1 watermark ranged from -12% (ie. a performance improvement) to 12.7% for insert opaque predicates and 4156% for exception branches. The performance implications for destroy the A2 watermark were similar with a 12% speed up for dynamic inliner and 15.8% and 4249% slowdowns for insert opaque predicates and exception branches respectively. Additionally, the A2 watermark could be destroyed with only a small degradation in performance using static inlining and method merging. Each resulted in only a 1% slowdown.

Both A1 and A2 are susceptible to additive attacks. Embedding an additional
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>A1</th>
<th>A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array folder</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Array splitter</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Block marker</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Budgeon signatures</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Boolean splitter</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Branch inverter</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Buggy code</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Class splitter</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Constant pool reorderer</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Duplicate registers</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Dynamic inliner</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Exception branches</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>False refactor</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Field assignment</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Inliner</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Insert opaque predicates</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Interleave methods</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Irreducibility</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Merge local integers</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Method merger</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Objectify</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Opaque branch insertion</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Overload names</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Parameter aliases</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Promote primitive registers</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Promote primitive types</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Publicizer</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Random dead code</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Rename registers</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Reorder parameters</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Split classes</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Static method bodies</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>String encoder</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Variable reassigner</td>
<td>☺</td>
<td>☺</td>
</tr>
</tbody>
</table>

⊕ : watermark found in all test applications
⊖ : watermark destroyed in at least one test application

Table 7.15: Results of applying the SANDMARK obfuscations on nine of the test applications which were watermarked using the Arboit algorithms.
watermark using either A1 or A2, regardless of which was used for the original embedding, has the effect of destroying the original as well as the new watermark. This occurs because the recognition procedure detects additional opaque predicates. So even though the original watermark was destroyed, the attacker will not be able to embed his own. Of the 11 watermarking algorithms presented in this chapter, only Graph Coloring-Based and Abstract Interpretation could not be used to launch an additive attack against the Arboit techniques.

It is also possible to launch successful collusive attacks against both A1 and A2. As with the Spread Spectrum algorithm, a collusive attack has the potential to reveal the opaque predicates contained in the library. This can lead to the construction of a class attack.

7.8 Collberg-Thomborson Algorithm

The first dynamic software watermarking algorithm was proposed by Collberg and Thomborson [35]. Instead of embedding the watermark directly in the code, code is added which builds the watermark as the program executes on a particular input. The basic idea is that the watermark is encoded in the structure of the graph which is built on the heap at runtime. Because the CT algorithm is dynamic, it requires annotation and tracing procedures in addition to embedding and recognition.

7.8.1 Embedding

The embedding process is when the watermark building code is actually added to the program. There are four distinct phases in the CT embedding process:

1. Convert the watermark into a graph.

2. Split the graph into several components.

3. Generate executable code which will build the components.

4. Replace annotation points with graph building code.
The first phase in embedding requires that the watermark be encoded as a graph. The particular graph structure used is unimportant as long as it is possible to convert between the graph representation and the watermark value. However, some graph structures are preferred due to properties such as high bit-rate, high resilience to attack, high stealth, etc. Collberg et al. [38] suggest five different graph types which would be suitable for encoding the watermark: permutation encoding, radix encoding, parent-pointer trees, reducible permutation graphs, and planted plane cubic trees.

In the second phase of embedding the graph is split into several components. This particular phase is not required, but potentially increases the stealth of the algorithm. For large watermark values, a large graph would be constructed. Concentrating this construction to a single location in the code could alert the attacker to the location. When splitting the graph three issues should be considered:

1. Each component should be basically the same size.
2. Each component should have a root node.
3. The number of edges between components should be minimized.

Based on the structure of the graph, executable code is generated in phase three. To aid in this step it is advantageous to first generate intermediate code. This step would be similar to the generation of intermediate code by a compiler prior to executable code generation. From the intermediate code, executable code can be generated for any target language. Additionally, semantics-preserving transformations, such as optimizations, can be applied to intermediate code prior to code generation.

In the final embedding phase the watermark building code is inserted in the program. The placement of the watermark code is based on the annotation points encountered during the tracing process. For each watermark component one annotation point is replaced. Any remaining annotation points are removed.
7.8.2 Recognition

As with embedding, recognition requires that the program is executed using the secret input sequence. When executed with the secret input sequence the same path will be traversed as during tracing. This will cause the watermark graph to be constructed on the heap. To observe the heap during execution the program has to be executed in an environment which enables monitoring, e.g., a debugger. It should be noted that in most cases the heap will contain a very large number of objects. However, the search space can be decreased by observing that the root node of the watermark graph will be one of the last objects added to the heap. Once the graph has been identified it is decoded yielding the watermark value.

7.8.3 Evaluation

Collberg et al. [38] provide the first description of a complete implementation of the CT algorithm for Java. In this work they use the SANDMARK framework to evaluate the algorithm with respect to stealth, data-rate, and resilience to attack. In this section we draw heavily on those results while performing further evaluation to enable a comparison with the other techniques.

Credibility The CT algorithm uses a blind recognizer to extract the watermark value from the program. Like many other algorithms of this type the CT algorithm is guaranteed to recover the watermark from a watermarked, but otherwise unaltered program.

Data-rate To embed a watermark using the CT algorithm the watermark is converted to a graph and graph building code is inserted in the program. This type of embedding technique usually enables any size watermark to be embedded in any application.

Through our evaluation we found that we were only able to embed a watermark in the application fft. This occurred because we were unable to gather a trace for the other applications which met the embedding criteria. Because we did not
have access to the source code for most of the test applications, we relied on the automated annotation feature. For the small applications the result was a trace without a single valid insertion point. For the larger applications like CaffeineMark and Java Grande the trace gathering exceeded our memory capabilities. Despite these limitations using fft we were able to embed watermarks of any size up to the maximum tried of 64 bytes.

**Stealth** Collberg et al. [38] define two types of stealth in their evaluation of the CT algorithm: *steganographic* and *local* stealth. They define an algorithm to have high steganographic stealth if given access to a watermarking algorithm and a watermarked program, the attacker cannot determine if the program was watermarked using the given algorithm. On the other hand, local stealth is defined such that if given access to a watermarking algorithm and a program which is known to be watermarked with that algorithm, the attacker cannot determine the location of the watermark within the program. The authors acknowledge that a watermarking technique may be stealthy for one host program but not another. To address this issue their measure of stealth is based on a universe of real programs. For an algorithm to have a high degree of stealth it must be stealthy for most of the program in the defined universe. Despite defining both steganographic and local stealth, Collberg et al. only evaluate the CT algorithm with respect to steganographic stealth.

To evaluate steganographic stealth, Collberg et al. [38] constructed a universe of 622 Java jar-files. For each jar-file a window of size 1-4 instructions was passed over the applications instruction sequence. The frequency of each particular instruction sequence was recorded. To simulate the watermarking process, watermark classes for uncycled Radix Graph watermarks of size 4-, 16-, 32-, and 64-bits were constructed. For each of these classes the instruction window frequencies were recorded. Finally, the window frequencies from the watermark classes were added to the frequencies for each application in the universe. An algorithm $A$ when embedding a $b$-bit watermark
Figure 7.19: Size impact incurred using the CT algorithm.

exhibits steganographic stealth for a program \( P \) if

\[
\frac{|\text{window types in a } b\text{-bit watermark but not in } P|}{|\text{window types that occur in } P|} < \delta
\]

Based on this definition they showed that when \( \delta = 0.1 \) a 32-bit watermark can be stealthily embedded in 63% of the applications in the universe.

**Overhead** Due to the difficulties associated with embedding watermarks in nine of the ten test applications, we were unable to perform a complete overhead evaluation. Using the \texttt{fft} application we embedded 32-, 64-, 128-, 256-, 512-bit watermarks. The graph in Figure 7.19 shows that the application size increased from 80% to approximated 700%. Because \texttt{fft} is one of the smaller test applications, a less significant impact would be expected for the larger applications such as \texttt{Java Grande}. For examples, if we assume that approximately the same number of bytes would be added to the \texttt{Java Grande} application for each of the watermark values, then the size increase would range from 1.3% to 11.2%. This increase would be considered acceptable for many applications.
**Robustness** Like the overhead evaluation, we were unable to perform a complete evaluation with respect to resistance against attack. To launch a subtractive attack against the CT algorithm requires the adversary to identify the graph building code. For many applications such a task will be quite feasible. Since the graph building code is not linked to the proper execution behavior of the program it can easily be removed with a bytecode editor. To counter this attack Thomborson et al. [97] devised a scheme for tamperproofing software watermarks that are embedded in dynamic data structures. The idea is to transform constant values in the watermark program into function calls which depend on the watermark data structure. This has the effect of linking the watermark construction with program execution, making the graph building code harder to remove.

The CT algorithm demonstrates a high level of resistance to distortive attacks. Of the 35 SANDMARK obfuscations, the algorithm is only vulnerable to the split classes obfuscation. However, this attack can be countered by using cycled graphs rather than plain graphs for the watermark. In our evaluation we watermarked the `fft` application using both cycled and plain graphs. When the split classes obfuscation was applied to the watermarked application which used a plain graph, the watermark was not recoverable. In the case of the cycled watermark graph, the watermark was recoverable.

Collberg et al. [38] note that the resistance to the split classes attack by using cycled graphs comes at the cost of a significantly reduced data-rate. This means a much larger cycle graph is required to encode the same watermark value in a plain graph, thus impacting both application size and performance. However, for less performance critical application the additional cost my be worthwhile.

The high resistance to distortive attacks only eliminates half of the threat associated with an additive attack. An adversary can still easily cast doubt on the original ownership by embedding a second watermark. Of the 11 watermarking algorithms presented in this chapter only the CT algorithm itself has the potential to destroy the original watermark. In each of the other cases, the original watermark is still recoverable but so is the second watermark, thus casting doubt on the original
ownership.

Like the other watermarking algorithms in this chapter, the CT algorithm remains vulnerable to the collusive attack. Because each watermark requires different graph building code, an attacker will be able to leverage these differences to successfully perform a collusive attack. Since the CT algorithm is highly resistant to distortive attacks, code obfuscation can be used to make the collusive attack more difficult.

7.9 Path-Based Watermarking

Path-based watermarking is an execution trace scheme in which the runtime branch structure of the program is manipulated to encode the watermark. The basis for the scheme is that the runtime branching structure is an inherent aspect of the program. However, the flexibility in manipulating the branch structure varies based on architectural restrictions. To address this issue, Collberg et al. [30] proposed two variations on the scheme. The first technique targets the features of Java bytecode while the second technique is aimed at watermarking the more flexible native code.

7.9.1 Java Version

In the Java scheme the watermark is encoded by modifying the sequence of branches which are taken or fall through on the secret input sequence. Because the watermark encoding is based on the dynamic behavior of the program, the technique includes a tracing phase which is used to aid in both embedding and recognition.

Tracing

To both embed and recognize a path-based watermark requires an execution trace of the program. For this particular technique the trace contains the sequence of basic blocks that were executed when the program was run on the secret input sequence. During the embedding phase the trace is used to identify appropriate locations for code insertion. In the recognition phase the trace is used to define a bit-string which
will be decoded into the watermark. The bit-string representation is a mapping between the branching behavior of the program and the binary representation of the watermark. A variety of different mappings can be constructed, however, most lack the necessary robustness. For example, given the branch “if \( P \) then \( Q \) else \( R \)' the bit-string representation could be if \( P \) is true 1, else 0. This representation lacks robustness in that an attacker can perform a branch inversion which will have the effect of reversing the bits. To make the mapping somewhat robust the following bit-string correspondence is defined:

- A 0-bit is obtained whenever the branch is immediately followed by the same instruction which followed the branch’s first occurrence.
- A 1-bit is obtained whenever the branch is immediately followed by a different instruction than that which followed the branch’s first occurrence.

Embedding

Prior to embedding the watermark it is split into a set of values using the Generalized Chinese Remainder Theorem and a block cipher. The splitting technique enables the watermarking scheme to be resistant to various semantics-preserving transformations by introducing redundancy and error correction. The watermark is embedded by inserting new branches such that the bit-string corresponding to the trace of basic blocks is the watermark bit-string. Two different techniques are presented by the authors for constructing the code sequences. In the loop technique a loop is generated which contains a conditional branch in its body. The supporting code is constructed so that the conditional branch will succeed or fail in the order of the bit-string for the particular watermark piece. The conditional code technique inserts a sequence of branches in a location which will be executed multiple times. To illustrate the Java version of the path-based watermark embedding consider the following code segment which computes the fibonacci of 3:
int m = 3;
int a = 0, b = 1;
for(int i=0; i < m; i++){
    int c = a + b;
    a = b;
    b = c;
}
print(b);

If we wanted to encode the watermark 0010 we would transform the code segment using the conditional code technique in the following way:

int m = 3;
int a = 0, b = 1;
int tmp = 0;
for(int i=0; i < m; i++){
    int c = a + b;
    a = b;
    b = c;
    if(a == 1) tmp++;
    if(b == c) tmp++;
}
print(b);

Recognition

To recognize the watermark an execution trace is captured using the secret input sequence. From the trace the bit-string representation is constructed. The bit-string is split into fixed sized blocks which are then decrypted using the same cipher used in embedding. Using the recovered watermark pieces and the generalized Chinese Remainder theorem the watermark is reassembled. Because various attacks could have been launched against the watermark, the authors suggest that an error correcting code is used when producing the watermark bit-string. This will make it possible to reassemble the string even when it was not possible to recover all of the watermark pieces.
Evaluation

Collberg et al. [30] implemented the Path-Based algorithm for Java using the SAND-Mark framework. In their research they evaluated the algorithm with respect to performance and size overhead and robustness against attack. In this section we provide our own evaluation and where possible compare our evaluation results to theirs.

Credibility The Path-Based algorithm uses a blind recognizer to extract the watermark value. Like many of the other algorithms in this chapter which use this type or recognition procedure, the Path-Based algorithm is guaranteed to recover the watermark from a watermarked, but otherwise unaltered program.

Data-rate To embed a watermark, the Path-Based algorithm adds new code to the program. This type of embedding technique enables any size watermark to be embedded in any application. However, a larger watermark requires that a greater number of conditional branches and/or loops be added to the program. For small applications such additions could become suspicious.

Stealth The main aspect of the Path-Based algorithm for Java which could compromise the stealth is the artificially-generated conditional branch and loop code segments. To encode a sequence of bits using the conditional branch technique requires the insertion of a sequence of conditional branches. For many applications such a sequence of instructions would be quite unusual.

Overhead To embed a watermark using the Path-Based algorithm first requires that a trace of the application is captured. During our evaluation we found that the even the trace of the small applications resulted in a file several Gigabytes in size. Because of the extensive trace needed for this algorithm and the limitations of our system we were only able to watermark three of the ten test applications: decode, fft, and CaffeineMark. We watermarked each of these applications with 32-, 64-,
128-, 256-, and 512-bit watermark. The graph in Figure 7.20 shows that impact on each of the applications changed very little even though the size of the watermark increased.

In the Path-Based algorithm the inserted code segments are actually executed in the watermarked application. Because of this we expected to observe a noticeable slowdown for CaffeineMark. However, when we embedded 32-, 64-, 128-, 256-, and 512-bit watermarks in the application the impact on performance was negligible. Table 7.16 shows the results of watermarking on the CaffeineMark application. Collberg et al. [30] noted a more significant slowdown using the CaffeineMark application, however, this degradation was observed when the number of watermark pieces was increased. For our experiment we used the default setting with respect to this variable.

Robustness It is assumed in the study of software watermarking algorithms that the attacker knows the algorithm used to embed the watermark. This along with the lack of stealth associated with the artificially-generated conditional branch and loop code segments makes it possible for an adversary to easily launch a successful
<table>
<thead>
<tr>
<th>Category</th>
<th>32-bit</th>
<th>64-bit</th>
<th>128-bit</th>
<th>256-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sieve</td>
<td>0.8%</td>
<td>1.6%</td>
<td>0.1%</td>
<td>2.6%</td>
<td>2.3%</td>
</tr>
<tr>
<td>Loop</td>
<td>-6.2%</td>
<td>-6.0%</td>
<td>-6.0%</td>
<td>-6.1%</td>
<td>2.1%</td>
</tr>
<tr>
<td>Logic</td>
<td>-0.1%</td>
<td>-0.4%</td>
<td>0.7%</td>
<td>-0.5%</td>
<td>-0.9%</td>
</tr>
<tr>
<td>String</td>
<td>1.8%</td>
<td>0.4%</td>
<td>1.6%</td>
<td>1.6%</td>
<td>1.8%</td>
</tr>
<tr>
<td>Float</td>
<td>1.4%</td>
<td>4.6%</td>
<td>-2.3%</td>
<td>-2.7%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Method</td>
<td>4.7%</td>
<td>1.2%</td>
<td>-12.1%</td>
<td>-12.6%</td>
<td>2.8%</td>
</tr>
<tr>
<td>Overall</td>
<td>0.4%</td>
<td>0.3%</td>
<td>-2.9%</td>
<td>-2.8%</td>
<td>1.6%</td>
</tr>
</tbody>
</table>

Table 7.16: CaffeineMark scores before and after embedding 32-, 64-, 128-, 256-, and 512-bit watermarks using the Path-Based algorithm.

subtractive attack. Using a bytecode viewer such as the one included in SandoMark an adversary can identify the location of successive conditional branch instructions. Since these instructions are not necessary for the proper execution of the program they can be removed using a bytecode editor.

Even though a subtractive attack can be launched with ease, the Path-Based algorithm for Java demonstrates a high level of resistance to distortive attacks. Of the 35 SandoMark obfuscations, the algorithm is only vulnerable to the opaque branch insertion obfuscation. This obfuscation inserts branches at random locations throughout the program. Such insertions have the potential to alter the bit sequence extracted from the program.

The implementation included in the SandoMark framework incorporates error correcting codes to improve robustness against the branch insertion attack. Using the CaffeineMark application, a 128-bit watermark, and the branch insertion obfuscation we were able to destroy the watermark by increasing the number of branches by 100%. However, this resulted in an overall slowdown of 65.8%. Collberg et al. [30] only observed a 50% slowdown when the number of branches was increased by 150%, but they do not indicate what test application they used.

The high resistance to distortive attacks only eliminates half of the threat associated with an additive attack. An adversary can still easily cast doubt on the original
ownership by embedding a second watermark. Of the 11 watermarking algorithms presented in this chapter only the Path-Based algorithm itself has the potential to destroy the original watermark. In each of the other cases, the original watermark is still recoverable but so is the second watermark, thus casting doubt on the original ownership.

Finally, the Path-Based algorithm, like most other watermarking algorithms, remains vulnerable to the collusive attack. By comparing two differently watermarked applications an attacker will be able to easily identify the inserted conditional branch and loop code segments. Collberg et al. [30] suggest the use of code obfuscation to address this vulnerability.

7.9.2 Native Code Version

The second path-based watermarking scheme proposed targets native executables. Many of the restrictions placed on Java bytecode do not exist with native executables. Because of this a different approach was used in embedding the watermark which relies on the use of a branch function. (For a detailed explanation of a branch function see Chapter 3.)

Tracing

As with the Java version, the watermark is encoded in the dynamic branch behavior or the program, thus, to aid in both embedding and recognition there is a defined tracing procedure. During embedding, the trace procedure identifies the sequence of unconditional (branch, target) pairs that are encountered when the program executes using the secret input sequence. To aid in recognition a finer-grain trace is required. In this case, it is not enough to record the unconditional (branch, target) pairs. We also need to know the address that was returned to after the function call completed. From this the branch function can be deduced.
Embedding

To embed the watermark, unconditional control flow edges are identified throughout the program, $b_i \rightarrow t_i$. Along each of the control flow edges a sequence of $k$ bits can be embedded by inserting additional calls to the branch function. Starting at $b_i$ a sequence of $k + 1$ branch function calls is constructed such that execution begins at $b_i$, continues through each of the additional $k$ branch function calls, and ends at $t_i$. The placement of the additional $k$ branch function calls encodes the watermark. Let $(a_0, a_1, ..., a_k)$ be the sequence of addresses encoding a $k$ bit watermark $w_0 w_1 \cdots w_{k-1}$ which begins at branch $b_i$ and ends at target $t_i$. The $k + 1$ branch function calls are inserted such that:

1. For each $a_i$, $0 < a_i \leq k$, the instruction immediately before $a_i$ is an unconditional jump.

2. if $w_i = 0$, $addr(a_{i-1}) > addr(a_i)$
   if $w_i = 1$, $addr(a_{i-1}) < addr(a_i)$

Using this scheme, the outcome of the branch instruction encodes a single bit of the watermark. A 0-bit is encoded through a backward jump and a 1-bit through a forward jump.

Figure 7.21 illustrates the native code version of the path-based watermark embedding. In this example the bit-string 01 is embedded in a code segment. Starting at $a_0$, the first bit 0 is encoded by a backward branch $a_0 \rightarrow a_1$, through a call to the branch function $f()$. The second bit, a 1, is encoded by a forward branch $a_1 \rightarrow a_2$, which is again routed through the branch function. Finally, control returns from $a_2$ to $t$ which is the end point of the watermark.

Recognition

To recognize the watermark the execution trace is captured. From this trace it is possible to identify the inserted branch function. For each instruction $a_i$ in the trace which calls the branch function, the corresponding $b_i$ is identified. Based on
the addresses of $a_i$ and $b_i$ the branch can be classified as either forward or backward. A 0-bit is recovered from a backward branch and 1-bit from a forward.

7.9.3 Evaluation

Collberg et al. [30] implemented the Path-Based algorithm for native code on top of PLTO, a binary rewriting tool for Intel IA-32 executables [43]. PLTO read in statically linked executables, dissembles them, and constructs a control flow graph. The CFG can be instrumented to extract an execution trace and then modified for watermark embedding. To evaluate the algorithm ten programs from the SPECint-2000 bench were used by the authors. In this section we summarize their evaluation with respect to cost (overhead) and resilience (robustness) and provide our own observations for the properties of credibility, data-rate, and stealth.

Credibility Like other algorithms which use an extraction based recognition procedure, the Path-Based algorithm is guaranteed to recognize a watermark in a watermarked but otherwise unaltered program. Because the algorithm relies on the use of a branch function there is the possibility of the algorithm producing false positives. This could occur when a program is obfuscated using a technique which
also relies on a branch function. Several such techniques have been proposed by
Linn and Debray [64].

**Data-rate** To embed a watermark, the Path-Based algorithm adds code to the
program. This type of embedding technique enables any size watermark to be
embedded in any application. However, a larger watermark will require a greater
number of calls to the branch function. For small applications this could be quite
costly.

**Stealth** The most critical aspect which compromises the stealth of the Path-Based
algorithm for native code is the repeated calling of the branch function. For each
bit of the watermark a single call to the branch function is added to the program,
thus, for a 128-bit watermark there will be 128 different calls to the branch function.
Such extensive reliance on a single function will be unusual for most programs. This
unusual statistic can be leveraged by an attacker to locate the watermark code and
eventually remove it.

**Overhead** To evaluate both the performance and size overhead incurred due to
the Path-Based algorithm, Collberg et al. watermarked each of the ten benchmark
programs with 128-, 256-, and 512-bit watermarks. In their evaluation they found
that the algorithm had only a modest impact on size. For a 128-bit watermark
the relative increase ranged from about 4% for crafty to about 15% for mcf. For
a 512-bit watermark the increase in size was only about an additional 1% for each
program.

For the performance evaluation the results were even better. The most significant
slowdown was observed when a 128-bit watermark was embedded in gap. In this
case the impact on performance was only about 4%. In fact, several programs
actually sped up by 2-3%. Overall, the mean slowdown ranged from -0.65% for
128-bit watermarks to 0.85 for 512-bit watermarks.
Robustness The aspect of the Path-Based algorithm which decreases the stealth also makes the algorithm highly susceptible to a subtractive attack. Through very simple static analysis an attacker can easily identify all calls to the branch function. Then using dynamic analysis the attacker can monitor the identified call instructions and the contents of memory. Through the monitoring the (branch, target) pairs can be identified as well as the effects that the branch function has on updating the contents of memory. Using this information the executable can be patched, thus, yielding a de-watermarked program.

The authors present an attack in which the branch is bypassed by replacing the calls to the branch function which a jump instruction of exactly the same size. The target of the jump is the address the branch function would transfer control to for that specific call instruction. This attack is a less sophisticated form of the subtractive attack we describe. Due to the branch function updating the contents of memory locations, the bypass may not succeed in all instances.

To simulate a distortive attack Collberg et al. used three different transformations: no-op insertion, branch sense inversion, and rerouting branch function entries. After each transformation the embedded watermark is still recoverable. Additionally, the use of the branch function provides minimal tamperproofing. Since a table is used to store displacements, any transformation which alters the displacement between a branch and target will cause the program to break. Because of this, applying the transformations no-op insertion and branch sense inversion yields obfuscated programs which do not function properly. Overall, because of the built in tamperproofing, the Path-Based algorithm for native code will be resistant to control flow obfuscations and many data obfuscations.

Collberg et al. also consider the effects of double watermarking, that is watermarking the program twice using the same algorithm. Like the no-op insertion and branch sense inversion transformations, double watermarking alters the code, thus yielding a non-functional program. This is the optimal outcome when combating the additive attack. Similar results can be expected for any additive attack which alters code addresses but does not update the mapping used by the branch function. If
the algorithms described in this chapter were all implemented for native code, only
the Graph Coloring-Based algorithm could be applied without necessarily yielding
a non-functional program. In this case both watermarks will be recoverable thus
casting doubt on the original ownership. Despite this one watermarking algorithm,
the Path-Based algorithm should have an overall high resistance to additive attacks.

7.10 Thread-Based Watermarking

Thread-based watermarking is a technique proposed by Nagra and Thomborson [78]
in which the watermark is embedded by manipulating the threading behavior of the
program. In general, the watermark is embedded by introducing additional threads
into a section of code which is single threaded. For example, in a program, assume
methods m1 and m2 are executed on a single thread. An additional thread could be
introduced, forcing the two methods to execute on separate threads. A watermark
can be encoded by altering which sections of code are executed on which threads.

7.10.1 Embedding

Prior to embedding the watermark it is necessary to obtain a trace of the program as
it is executed with the secret input sequence. The trace provides information as to
which basic block is being executed on which thread, i.e. the result of the trace is a
series of tuples \((B_i, T_i)\) where \(B_i\) is the block id and \(T_i\) is the thread id. Each byte of
the watermark is embedded by randomly selecting a thread \(T_i\) and a subsequence of
\(n\) unique basic blocks, in execution order, which were executed on \(T_i\). In the authors’
implementation, a 0 bit is encoded by executing a sequence of 3 basic blocks on 3
different threads. A 1 bit is encoded by executing the first and third blocks, in a
3 block sequence, on one thread and the second block on a different thread. This
particular encoding scheme is illustrated in Figure 7.22. In this example, blocks A,
B, and C, in the unwatermarked program P, are executed on a single thread. To
embed a 0-bit, each of the blocks are executed on individual threads. On the other
hand, a 1-bit is embedded by forcing the execution of blocks A and B on a single
thread and block B on a different thread. Notice that this encoding scheme does not tie a specific block to a specific thread, thus making the scheme more resilient to attacks. However, alternate encoding schemes can be conceived.

7.10.2 Recognition

Recognition of the watermark is accomplished by executing the watermarked program using the secret input sequence and observing the threading behavior. As in embedding, the first step in extracting the watermark is to obtain an execution trace based on the secret input. This will yield a sequence of tuples $(B_i, T_i)$. However, in recognition the focus is on the transition between threads. A list of thread IDs is constructed such that the thread IDs are in basic block order. An id is only recorded if two consecutive blocks are executed on different threads. To reconstruct the watermark every possible subsequence of 3 distinct thread IDs, obtained from the trace, are constructed. From the subsequences all possible watermarks are constructed. The watermark is verified as legal through look-up in an external table T. The additional information required during recognition indicates that this algorithm uses an informed recognition procedure. Additionally, the recognition procedure is classified as a detector because it reports the presence of a watermark instead of
extracting the watermark. This yields a recognizer with the following signature:

\[
\text{recognize}(P', k, T) \rightarrow \mathbb{B}
\]

7.10.3 Evaluation

Nagra and Thomborson implemented the Thread-Based algorithm for watermarking Java bytecode, however, we have yet to include an implementation within SAND-MARK. Since we do not have an implementation, the evaluation presented in this section is based on a summary of their evaluation and our own observations.

Credibility The Thread-Based algorithm uses an informed recognition procedure which reports true for false. During recognition all possible watermark sequences are constructed. Through table look-up the watermarks are verified. If a watermark is found in the external table then the recognizer reports true, thus indicating that the application contains a valid watermark. With this type of recognition procedure there is the possibility of false detections. However, based on the analysis by Nagra and Thomborson, the rate of false detections should be quite small. They estimate that when using a table composed of $2^{24}$ valid 48-bit codes, the false detection rate is less than $100/2^{24}$ for the benchmark JFig.

Data-rate The data-rate of the Thread-Based algorithm is based on the number of distinct basic blocks identified during the tracing process. To encode a single bit requires a sequence of three basic blocks, thus to encode even a single byte requires the identification of at least 24 distinct basic blocks. On initial inspection it may appear that such a requirement would prevent the embedding of larger watermarks.

Nagra and Thomborson failed to provide an experimental evaluation of the data-rate of the Thread-Based algorithm, however, based on our theoretical analysis of the algorithm we can make a reasonable approximation. In our analysis of 1132 Java applications we found that the average application has three packages, the average package has 11 classes, the average class has nine methods, and the average
method has 16 basic blocks. Based on these numbers we estimate that the average application contains approximately 4752 basic blocks. If we had an input sequence that executed a third of these we could embed 512-bits. Therefore, even though the size of the watermark is restricted by the size of the application, the Thread-Based algorithm should be able to embed a reasonable size watermark in the average program.

**Stealth** The embedding of a Thread-Based watermark relies on the introduction of new threads and new thread transitions. When the authors embedded a 48-bit watermark in the JFig application the thread count went from seven to twenty-five, that is three additional threads for each byte of the watermark. Based on this rate of thread addition, 192 new threads would be introduced for a 512-bit watermark. Such a statistic could be unusual for many applications and may alert an attacker that this particular algorithm was used for watermarking.

**Overhead** To evaluate the size and performance impact incurred as a result of Thread-Based watermarking, Nagra and Thomborson use three testcases: TTT, a GUI based tic-tac-toe program; JFIG, a GUI based figure editor; and SciMark, a composite Java benchmark used to evaluate the performance of scientific and engineering applications. In evaluating the size overhead they found that for each bit embedded the application size increased by 1.2 KB. For small applications such an increase in size will be significant. If we were to hypothetically embed a 32-bit watermark in each one of our 10 test applications (and it is unclear if this could actually be done for all of the applications) we would see an increase in size that ranged from 20% to almost 1400%. For the application decode and puzzle which should be approximately the size of TTT an increase in size of 607% and 446% respectively should be expected. Java Grande, which is the largest application, would have an expected increase in size of 20%. Table 7.17 shows the hypothetical size impact of the Thread-Based algorithm.

Thread switching code can be very costly in terms of execution time. Because
<table>
<thead>
<tr>
<th>Application</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>607%</td>
</tr>
<tr>
<td>fft</td>
<td>1197%</td>
</tr>
<tr>
<td>illness</td>
<td>311%</td>
</tr>
<tr>
<td>lu</td>
<td>1365%</td>
</tr>
<tr>
<td>machineSim</td>
<td>217%</td>
</tr>
<tr>
<td>matrix</td>
<td>1267%</td>
</tr>
<tr>
<td>probe</td>
<td>1388%</td>
</tr>
<tr>
<td>puzzle</td>
<td>446%</td>
</tr>
<tr>
<td>Java Grande</td>
<td>20%</td>
</tr>
<tr>
<td>CaffeineMark</td>
<td>406%</td>
</tr>
</tbody>
</table>

Table 7.17: Hypothetical size impact incurred using the Thread-Based algorithm.

of this we would expect to see a noticeable slowdown in applications watermarked using the Thread-Based algorithm. Two of the test programs used by Nagra and Thomborson are GUI program which wait for user interaction and have few time critical loops. For the JFig application very little slowdown was observed for a 48-bit watermark. A greater slowdown was observed for TTT using the same watermark with the slowdown being almost a factor of 2 or approximately 90%. The most significant slowdown was observed for SciMark. For this application the slowdown was approximately a factor of 8 or 700%. Since this application is similar in nature to CaffeineMark and Java Grande we would expect similar results for these two test applications. Based on these results we hypothesize that the Thread-Based algorithm would not be a good choice for watermarking performance critical applications.

**Robustness** In the study of software watermarking we assume the attacker has knowledge about each of the algorithms such as the embedding algorithm. The Thread-Based algorithm introduces a high number of new threads to an application. Knowing this an attack will easily be able to identify applications which have been watermarked using the Thread-Based algorithm through simple analysis techniques. Once the watermark technique is known an attacker can launch a directed attack. Nagra and Thomborson acknowledge that such an attack could be carried out by simply inserting `Thread.yield()` inside a based block. Blindly applied, such an
attack could have a significant performance impact.

The recognition procedure used by the Thread-Based algorithm relies on the execution behavior. Because of this it is resistant to the traditional layout, data, and control flow obfuscations present in SANDMARK. An obfuscation could be constructed based on the Thread.yield() idea, but this transformation would probably have a performance impact similar to the exception branch obfuscation.

Despite the high resistance to distortive attacks an adversary could cast doubt on the original ownership through an additive attack. If an additive attack is launched using any one of the other algorithms presented in this chapter, the original mark will still be recoverable. However, the second watermark should also be recoverable. Additionally, based on our analysis it seems possible that an additive attack which used the Thread-Based algorithm could damage the original watermark. However, such a hypothesis would need to be experimentally tested. Overall, the Thread-Based algorithm is moderately resistant to additive attacks since embedding the second watermark will not destroy the original in almost all cases.

7.11 Abstract Interpretation Watermarking

A watermarking transformation which is based on abstract interpretation embeds the watermark in such a way that it can only be extracted by analyzing the concrete semantics of the code. Currently only a single abstract watermarking technique has been proposed. Cousot and Cousot [40] proposed a technique in which the watermark is embedded in such a manner that extraction can only be accomplished by statically analyzing the concrete semantics of the code using abstract interpretation.

7.11.1 Embedding

The embedding of the watermark value is accomplished by embedding a set of values which can later be used to reconstruct the watermark through the use of the Chinese Remainder Theorem.
Theorem 1 (Chinese Remainder Theorem). Suppose that $m_1, m_2, \ldots, m_n$ are pair-
wise relatively prime positive integers, and let $a_1, a_2, \ldots, a_n$ be integers. Then the
system of congruences $x \equiv a_i \pmod{m_i}$ for $1 \leq i \leq n$, has a unique solution modulo
$M = m_1m_2\ldots m_n$ which is given by

$$x = a_1M_1y_1 + a_2M_2y_2 + \cdots + a_nM_ny_n$$

where $M_i = \frac{M}{m_i}$ and $y_i = (M_i)^{-1} \pmod{m_i}$ for $1 \leq i \leq n$.

The Chinese Remainder Theorem makes it possible to solve a system of congru-
ences but only in limited circumstances, thus, the values used in embedding must
be of a specific form. To begin the watermarking process a sequence $n_1, \ldots, n_k$ of
pairwise relatively prime positive integers are chosen. These integers should remain
secret and can be viewed as the secret key. The watermark value $w$ is chosen such
that $w \in [0, n)$ where $n = \prod_{i=1}^{k} n_i$. For each $n_i$, compute $w_i \in [0, n_i)$ such that
$w \equiv w_i \pmod{n_i}$. This yields the set of values $V = \{(w_1, n_1), \ldots, (w_k, n_k)\}$ which
are used in the embedding.

For each of the value pairs in the set $V$, a method containing a loop is chosen
from the program to be used in embedding. A new variable $W$ is added to the
method with an initial value $P(1) \equiv w_i \pmod{n_i}$. In the body of the loop the value
of $W$ is updated such that $W = Q(W)$ where $Q(W) \equiv w \pmod{n_i}$.

This particular embedding process is simplistic and does not hide the value of
$w_i$. To this end, an obfuscated embedding process was proposed. In this version $W$
is initialized using a polynomial of the form:

$$P(x) = x^2 + h_1 x + h_0 = (x + h_1)x + h_0$$

with coefficients $h_0 = 2w_i + r_1n_i$ and $h_1 = -(1 + w_i) + r_2n_i$ where $r_1$ and $r_2$ are
random integers. This makes it possible to spread the initialization of $W$ over four
steps:
\[ W = 1; \]
\[ T = W + h_1; \]
\[ T = W \ast T; \]
\[ W = T + h_0; \]

where \( T \) is a new temporary variable inserted in the program. To further obfuscate the initialization, the steps can be dispersed throughout the method.

A similar technique can be used in the body of the loop where the value of \( W \) is updated, \( W = Q(W) \). Again \( Q \) is chosen to be a polynomial, of the form:

\[ Q(x) = r_3 x^2 + r_4 x + c = (r_3 x + r_4) x + c \]

where \( r_3 \) and \( r_4 \) are random integers and \( c \) is chosen to ensure that \( w_i = Q(w_i) \):

\[ c = w_i - (r_4 w_i + r_3 w_i^2) \]

Thus \( W \) is updated through a sequence of steps:

\[ T = W \ast r_3; \]
\[ T = T + r_4; \]
\[ T = T \ast W; \]
\[ W = T + c; \]

The last step is only required when \( c \neq 0 \).

To illustrate this watermarking technique suppose we want to watermark the simple method in Figure 7.23 (a) which calculates the mth fibonacci number. We choose our secret key to be \( n_1 = 17, n_2 = 21 \) and \( w = 151 \). Using the procedure to split the watermark we compute the pairs \((15, 17)\) and \((4, 21)\) to be used in embedding. To embed the first pair, \((15, 17)\), we choose the four random values \( r_1, r_2, r_3, r_4 \) to be 2, 5, 4, 7 respectively. Initialization of \( W \):
\[ h_0 = 2w_1 + r_1n_1 = 2(15) + 2(17) = 64 \]
\[ h_1 = -(1 + w_1) + r_2n_1 = -(1 + 15) + 5(17) = 59 \]
\[ W = 1 \]
\[ T = W + 59 \]
\[ T = W \times T \]
\[ W = T + 64 \]

Update of \( W \) in the loop:

\[ c = w_1 - (r_4w_1 + r_3w_1^2) = 15 - [7(15) + 4(15^2)] = -990 \]
\[ T = W \times 4 \]
\[ T = T + 7 \]
\[ T = T \times W \]
\[ W = T - 990 \]

The code transformation associated with embedding (15,17) is illustrated in Figure 7.23 (b) with the variable \( d \) representing \( W \) and \( e \) representing \( T \).

To embed the second pair, (4,21), we repeat the process choosing the four random values to be 9, 3, 7, 6. Initialization of \( W \):

\[ h_0 = 2w_2 + r_1n_2 = 2(4) + 9(4) = 44 \]
\[ h_1 = -(1 + w_2) + r_2n_2 = -(1 + 4) + 3(21) = 58 \]
\[ W = 1 \]
\[ T = W + 58 \]
\[ T = W \times T \]
\[ W = T + 44 \]

Update of \( W \) in the loop:
\[ c = w_2 - (r_4 w_2 + r_3 w_2^2) = 4 - [6(4) + 7(4^2)] = -132 \]

\[ T = W \times 7 \]

\[ T = T + 6 \]

\[ T = T \times W \]

\[ W = T - 132 \]

This embedding is illustrated in Figure 7.23 (c) with the variable \( f \) representing \( W \) and \( g \) representing \( T \).

### 7.11.2 Recognition

Watermark recognition requires that the program is statically analyzed using abstract interpretation and the pairwise relatively prime positive integers \( n_1, \ldots, n_k \). During the analysis the variables are examined to find those whose values are a constant \( \text{mod } n_i \). The constant value is \( w_i \). Once all possible \( (w_i, n_i) \) pairs are recovered the watermark can be reconstructed using the Chinese Remainder Theorem. Because the technique relies on the Chinese Remainder Theorem to reconstruct the watermark, it is possible to recover the watermark even if only a portion of the program code is present. This has the advantage of being able to detect theft of a portion of the program such as a library or still being able to detect theft when the program has been altered.

### 7.11.3 Evaluation

Cousot and Cousot implemented the Abstract Interpretation algorithm for Java using the Soot optimization framework [99], however, we have yet to include an implementation within SANDMARK. Since we do not have an implementation of the algorithm and their evaluation was not available for us to analyze, the evaluation presented in this section is based on the evaluation by Cousot and Cousot and our own observations.
Figure 7.23: Illustration of the code transformation as a result of using abstract-interpretation based software watermarking.
Credibility In order to recognize the watermark, the pairwise relatively prime integers \( n_1, \ldots, n_k \) are required. Despite this fact, the recognition procedure can still be implemented as a blind recognizer which extracts the watermark by obtaining the values from the secret key. Based on this type of recognition procedure it is our belief that the Abstract Interpretation algorithm can be constructed such that it is guaranteed to recognize a watermark in a watermarked but otherwise unaltered program. In fact, Cousot and Cousot provide a correctness theorem, but no proof, for signature extraction.

The recognition technique relies on the Chinese Remainder Theorem to reconstruct the watermark. Such a technique makes it possible to recover a watermark from only a portion of the program code. Because of this, it seems possible that the algorithm could be forced into producing false positives. However, like most of the other algorithms in which a false positive is possible, we believe the false positive rate will be very low. Overall, the Abstract Interpretation algorithm should demonstrate a very high level of credibility.

Data-rate To embed a watermark, the Abstract Interpretation algorithm adds code to the program. This type of embedding technique enables the embedding of any size watermark in any application which contains at least one loop. A larger watermark value will require a larger set of value \( V \). For each pair in the set of value \( V \) at least two variable declarations and six arithmetic expressions are added, thus a larger watermark requires more extensive code additions. For small applications such code additions could become suspicious.

Stealth There is at least one aspect of the Abstract Interpretation algorithm which could severely compromise the stealth. As can be seen in Figure 7.23, the embedding process results in the insertion of unusual integer literal constants. As the watermark value increases, the size of the literal constants will also increase. Based on an analysis of Java bytecode programs we conducted in Collberg et al. [34], 93% of all integers found in programs are less than 1000, thus, the occurrence of large literal constants
is quite rare. Additionally, we found that 63% of all literal integers are zero, powers of two, or powers of two plus/minus one. The results from this study indicate that in real Java programs most constants are small or close to a power of two, thus the Abstract Interpretation algorithm is likely to have low stealth.

**Overhead** In the evaluation conducted by Cousot and Cousot there is no mention as to the size impact incurred by the Abstract Interpretation algorithm. Due to the lack of an implementation we cannot comment on the exact impact this algorithm would have. However, we can speculate based on the type of embedding technique used and our hand watermarked example from Figure 7.23. In this example we embedded a three digit watermark which was split into two parts. The additional code required to embed the watermark increased the size of our application by 169 bytes. An increase of approximately 100 bytes per watermark part is quite minimal for medium to large applications.

Cousot and Cousot claim that the impact on performance for medium and large programs is negligible. They state that there were no observable changes in the required memory or computation time. While it does seem reasonable that the algorithm would only have a minimal impact on medium and large programs when a small watermark is embedded, no experimental results were disclosed in their evaluation. Based on our understanding of the algorithm, a larger watermark could require more extensive code additions. Since these code additions are executed it seems reasonable to conclude that a larger watermark would have a larger impact on performance.

**Robustness** The lack of stealth leaves the Abstract Interpretation algorithm vulnerable to a variety of attacks based on static and dynamic analysis. For example, an attacker could use static analysis to identify all of the large or unusual literal constants. Once the sections of code, which could be a small set, are identified, a concentrated obfuscation attack could be employed. Or dynamic analysis can be used to monitor program variables which are updated inside of loops. Those vari-
ables which appear to have random values may be part of the watermark. Once identified a more concentrated attack can be used to remove or obscure the values. For each of these attacks, the authors suggest counter-attacks. These include deriving large constants from smaller ones or using techniques to prevent the watermark code from being executed frequently. Such techniques will not prevent recognition but will make attacks based on static or dynamic analysis more difficult.

With respect to distortive attacks, Cousot and Cousot tested the strength of the Abstract Interpretation algorithm using the obfuscators JCloak [8] and Zelix Klassmaster [11]. Both obfuscators provide name obfuscation, some type of control flow obfuscation, and string encryption. In their experiment a single method in a class was watermarked and then the class was obfuscated. In each case the watermark was successfully recovered.

Cousot and Cousot also provide a theoretical analysis of the strength of the algorithm based on various categories of control flow and data obfuscations. These include sequential code reordering, conversion of static analysis to procedural data, outlining, parallelization, globalization, and variable splitting and merging. For each of these a method is described for creating an abstract analyzer which could recognize the watermark even in the presence of such an obfuscation. However, it is unclear if a single analyzer could be created that would combat all of the attacks simultaneously. To answer such a question, further implementation and evaluation is required.

Depending on how the Abstract Interpretation algorithm is implemented it has the potential to be partially resistant to an additive attack launched using the algorithm presented in this chapter. Again, it is unclear if a single analyzer could be created which would simultaneously combat all of the attacks. The algorithm is only partially resistant because the original proof of ownership will be recoverable but so will the second watermark. This, of course, will cast doubt on the original ownership. The algorithm is also resistant to double watermarking using the same algorithm as long as a unique key is used for each watermark.

Finally, based on our analysis, the Abstract Interpretation algorithm appears to
Figure 7.24: Two identical methods which were differently watermarked using the Abstract Interpretation algorithm. Static analysis can be used to launch a collusive attack.

be highly vulnerable to collusive attacks. For example, consider the two differently watermarked methods in Figure 7.24. Through very simple static analysis or a tool such as the SANDMARK Diff Tool, the differences can be identified, thus, identifying the watermark code. Once identified, the watermark can easily be removed. To combat this attack Cousot and Cousot rely on what has become the customary solution of using obfuscation to make the programs differ everywhere. This is a viable option, but may not always be feasible due to size and/or performance overhead incurred. Based on the evaluations in this chapter it can be seen that some of the obfuscations have a significant impact on performance.

7.12 Discussion

SANDMARK includes an implementation for nine of 11 algorithms presented in this chapter. Thread-Based and Abstract Interpretation are the only algorithms not included in the SANDMARK framework. For each of the nine algorithms we performed
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Credibility</th>
<th>Data-rate</th>
<th>Overhead</th>
<th>Resistance to Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Size</td>
<td>Performance</td>
</tr>
<tr>
<td>DM</td>
<td>high</td>
<td>usable</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Moskowitz-Cooperman</td>
<td>high</td>
<td>exceptional</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Graph Coloring-Based</td>
<td>low</td>
<td>usable</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Spread Spectrum</td>
<td>medium</td>
<td>exceptional</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Dummy Method</td>
<td>high</td>
<td>exceptional</td>
<td>extreme</td>
<td>high</td>
</tr>
<tr>
<td>GTW_{SM}</td>
<td>high</td>
<td>exceptional</td>
<td>extreme</td>
<td>high</td>
</tr>
<tr>
<td>AT</td>
<td>high</td>
<td>exceptional</td>
<td>low</td>
<td>moderate</td>
</tr>
<tr>
<td>AZ</td>
<td>high</td>
<td>exceptional</td>
<td>moderate</td>
<td>moderate</td>
</tr>
<tr>
<td>CT</td>
<td>high</td>
<td>exceptional</td>
<td>moderate</td>
<td>NA</td>
</tr>
<tr>
<td>PBW</td>
<td>high</td>
<td>exceptional</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>

Table 7.18: A comparison of nine of the software watermarking algorithms presented in this chapter.

A thorough evaluation. In this section we draw on those evaluations to provide an overall comparison. Our comparison is based on those properties for which we have a quantitative measure. These include credibility, data-rate, size and performance overhead, and resistance to distortive and additive attacks. Table 7.18 provides a summary of the evaluation results for the nine algorithms.

Of the nine algorithms, only two do not demonstrate a high level of credibility: Graph Coloring-Based and Spread Spectrum. The Graph Coloring-Based algorithm has a low credibility due to a few errors in the algorithm. This will be explored in the next chapter. The credibility of the Spread Spectrum algorithm suffers due to the type of recognizer used.

Like the credibility property, only two algorithms have a data-rate classification which is less than high: DM and Graph Coloring-Based. Both of these algorithms are based on a reordering transformation. This type of embedding transformation generally makes use of very few new instruction to embed the watermark. Instead of inserting code already existing code is simply manipulated. Such a transformation will obviously restrict the data-rate. Despite this fact, a usable data-rate was achieved for at least one of the test applications for each of DM and Graph Coloring-Based.

With respect to the overhead property, only one algorithm rates poorly in both size and performance impact: GTW_{SM}. This algorithm had an extreme impact on
the size of the test applications and significantly impacted the performance. The only other notable algorithm was the Moskowitz-Cooperman algorithm. Because of the need to add a custom class loader, this algorithm had a high impact on the size of the test applications.

Finally, three of the algorithms in this chapter demonstrate a high resistance to distortive attacks: Moskowitz-Cooperman, CT, PBW. The Moskowitz-Cooperman algorithm embeds the watermark in an image in the program. Because of this it is naturally resistant to the common distortive and additive attacks used to evaluate the strength of a software watermarking algorithm. However, the algorithm also incorporates a tamperproofing mechanism to protect against common distortive attacks aimed at media watermarking. Due to the dynamic nature of the CT and PBW algorithms they are more naturally resistant to attack.
CHAPTER 8

ALGORITHM EXTENSIONS

In this chapter we present the results of our detailed examination of two software watermarking algorithms. In this work we attempt to improve on the algorithms by making modifications which maintain the essence of the original techniques. In the Graph Coloring-Based algorithm we focus on improving recognition reliability. With the Arboit techniques we attempt to improve robustness by converting the techniques from static to dynamic.

8.1 Graph Coloring-Based Watermarking

Graph Coloring-Based watermarking was originally proposed as a technique for watermarking FPGA designs [59]. However, the authors suggested that by uniquely altering the graph used by a graph coloring register allocator the technique could be applied to watermarking software. Unfortunately, the Graph Coloring-Based algorithm contains significant flaws when directly applied to software watermarking. In this section we provide a thorough explanation of the shortcomings of the algorithm as well as the modifications necessary to make it a viable. Much of the work in this section was presented in a previous publication [73]. Despite the difficulties associated with apply the Graph Coloring-Based algorithm to software watermarking the technique remains interesting due to the high degree of stealth and the applicability within other watermarking domains.

8.1.1 Algorithmic Shortcomings

As we studied the application of the Graph Coloring-Based algorithm to software watermarking we discovered the embedding algorithm contained a single minor flaw
and the recognition algorithm contained two, more problematic flaws. In the embedding algorithm we made the observation that it may not be possible to embed a bit for every vertex in the graph. This flaw can be corrected by making a very simple modification to the embedding algorithm: use two separate variables to maintain the index of the watermark bit and the current vertex. Based on this modification the new embedding algorithm becomes:

**Algorithm Embed**(program, watermark):
1. $G(V, E) \leftarrow$ interference graph for program;
2. $V \leftarrow \text{order}(V)$;
3. $G'(V', E') \leftarrow$ copy of $G(V, E)$;
4. $j \leftarrow 0$;
5. for each $v_i \in V$ do
6. if possible choose $v_{i_1}, v_{i_2}$ such that $(v_{i_1}, v_{i_1}), (v_{i_2}, v_{i_2}) \notin E$ and $i_2 > i_1 > i$ (mod n) and $(v_i, v_j) \in E \forall i < j < i_1, i_1 < j < i_2$ (mod n) then
7. if $w_i == 0$ then
8. add edge $(v_i, v_{i_1})$ to $E'$;
9. else
10. add edge $(v_i, v_{i_2})$ to $E'$;
11. $j ++$;
12. performRegisterAllocation($G'$);

The first flaw in the recognition algorithm involves extracting the wrong bit, e.g. we recognize a 1 bit when we embedded a 0. To illustrate this error consider the graphs in Fig. 8.1(a) and 8.1(b). We wish to embed the message $M = 00$ in the graph in Fig. 8.1(a). By following the above algorithm, bit $m_1$ is embedded by choosing $v_1 = V1, v_{i_1} = V3, v_{i_2} = V4$ and adding an edge between vertices $V1$ and $V3$. Bit $m_2$ is embedded by choosing $v_2 = V4, v_{i_1} = V1, v_{i_2} = V2$ and adding an edge between vertices $V4$ and $V1$. The new interference graph as well as the new coloring can be seen in Fig. 8.1(b). When the original recognition algorithm is used, the first set of vertices selected are $V1$ and $V3$. Since there are no vertices between
V1 and V3 that are not connected to V1, the recovered bit is a 0. The second set is V1 and V4. There is one vertex between V1 and V4 that is not connected to V1, so the recovered bit is 1. Thus, we recognize the message 01 even though the message 00 was embedded.

The second recognition flaw occurs in instances where more bits are recovered than where embedded. For example, when we embed the message 101 in the graph in Fig. 8.2(a) we obtain the graph in Fig. 8.2(b). By following the recognition algorithm we recover the message 1001. In fact, we discovered that as the number of vertices in the graph increases so does the number of extra bits recovered.

The cause of both of the problems can be traced back to a single assumption made about the coloring of the vertices: any two unconnected vertices are colored differently if and only if an edge is added between them during embedding. This assumption turns out to be incorrect as illustrated in Figure 8.2. When we embed the message 101 it forces vertices V3 and V4 to be colored differently even though no edge is added between them. The assumption fails because any particular vertex could be included in multiple triples. In this context, a triple is a set of three vertices in which no two vertices in the set are connected by an edge. For example, the vertices V1, V3, and V4 in Figure 8.2(a) form a triple. Thus, even if an edge is not added in one selection it could be in another. The unpredictability is also influenced by vertices in a triple that are not all of the same color. When this occurs it is possible that the color of the vertices could be influenced by the coloring of an
adjacent vertex which is not part of the triple.

**Definition 10** (triple). Given a graph $G = (V, E)$, a set of 3 vertices $\{v_1, v_2, v_3\}$ is considered a triple if

1. $v_1, v_2, v_3 \in V$, and
2. $(v_1, v_2), (v_1, v_3), (v_2, v_3) \not\in E$

In software watermarking, accurate recovery is crucial in providing proof of ownership. The details necessary to make the **Graph Coloring-Based** technique a viable algorithm are explained in the next section.

### 8.1.2 Algorithmic Improvements

To eliminate the unpredictability during recognition we devised an embedding algorithm which places additional constraints on which vertices can be selected for a triple. We call these triples **colored triples**. (Note that condition 3 implies condition 2 but remains to emphasize the additional constraint.)
Definition 11 (colored triple). Given an $n$-colorable graph $G = (V, E)$, a set of three vertices \{v_1, v_2, v_3\} is considered a colored triple if

1. $v_1, v_2, v_3 \in V$,

2. $(v_1, v_2), (v_1, v_3), (v_2, v_3) \not\in E$, and

3. $v_1, v_2, v_3$ are all colored the same color.

The key idea is to select the triples so that they are isolated units that will not affect the coloring of other vertices in the graph. The embedding is accomplished using the algorithm below in which a colored triple is selected and an edge is added based on the bit value. In addition, we use a specially designed register allocator which only changes the coloring of one of the two vertices involved in the added edge and no other vertices in the graph.

Algorithm Embed\(\text{program, watermark}\):

1. $G(V, E) \leftarrow$ interference graph for \text{program};
2. $V \leftarrow \text{order}(V)$;
3. $G'(V', E') \leftarrow$ copy of $G(V, E)$;
4. $j \leftarrow 0$;
5. for each $v_i \in V$ which has not already been selected for a triple do

6. \hspace{1em} if possible choose $v_{i_1}, v_{i_2} \in V$ such that $v_i, v_{i_1}, v_{i_2}$ form a colored triple in $G$ AND $i_2 > i_1 > i$ (mod $n$) AND $(v_i, v_{j_1}) \in E$; \forall $i < j < i_1, i_1 < j < i_2$ (mod $n$) AND $v_{i_1}, v_{i_2}$ have not already been selected for a triple then

7. \hspace{1em} \hspace{1em} if $w_j == 0$ then

8. \hspace{1em} \hspace{1em} add edge $(v_i, v_{i_1})$ to $E'$;

9. \hspace{1em} \hspace{1em} else

10. \hspace{1em} \hspace{1em} add edge $(v_i, v_{i_2})$ to $E'$;

11. \hspace{1em} $j + +$;

12. performRegisterAllocation($G'$);

The enhanced recognition algorithm is slightly more complex than the original algorithm and relies on additional information to recover the watermark. For recog-
inition, two different interference graphs are required, the watermarked graph and the original graph. Recognition is accomplished by identifying the triples as they would have been selected during the embedding phase. Once a triple \((v_i, v_i', v_i'')\) has been identified that same triple is examined in the embedded coloring \((v_i', v_i', v_i'')\). If \(v_i'\) and \(v_i''\) are different colors then we know that a 0 was embedded, otherwise a 1 was embedded.

**Algorithm** Recognize\((\text{program})\):

1. \(G'(V', E') \leftarrow\) watermarked interference graph for \(\text{program}\);
2. \(G(V, E) \leftarrow\) original interference graph for \(\text{program}\);
3. \(V' \leftarrow\) order\((V)\);
4. \(V \leftarrow\) order\((V)\);
5. **for** each \(v_i \in V\) which has not already been selected for a triple **do**
6. **if** possible choose \(v_{i_1}, v_{i_2} \in V\) such that \(v_i, v_{i_1}, v_{i_2}\) form a colored triple in \(G\) AND \(i_2 > i_1 > i\) (mod \(n\)) AND \((v_i, v_j) \in E; \forall i < j < i_1, i_1 < j < i_2\) (mod \(n\)) AND \(v_{i_1}, v_{i_2}\) have not already been selected for a triple **then**
7. **if** \(v_i'\) and \(v_i''\) are different colors **then**
8. \(\text{wm.append}^{(\text{"0"}^n)}\);
9. **else** \(\text{wm.append}^{(\text{"1"}^n)}\);
10. **return** \(\text{wm}\);

Upon initial inspection it appears that an informed recognizer is required to extract the watermark. It is true that the original coloring is required but this can be obtained by running the graph coloring register allocation algorithm over the interference graph obtained from the watermarked graph. Since the variable interferences were artificially imposed they do not actually appear in the watermarked graph which allows us to obtain the original coloring.

Figure 8.3 illustrates how an interference graph is modified to embed a 0-valued bit using the graph based watermarking technique. By adding an artificially imposed constraint between vertices 2 and 3, which represents a 0, a new coloring is imposed on vertex 3 thereby resulting in a new register allocation. In this example only a
single bit can be embedded in the graph. For recognition to be reliable, the selected triples must be isolated units. Since vertices 3 and 4 were selected for the initial triple, we are unable to construct a second isolated triple.

### 8.1.3 Evaluation

In a previous publication we provided an evaluation of the Graph Coloring-Based$_{SM}$ algorithm. In this section we provide updated results based on the more extensive attack capabilities currently included in SANDMARK.

**Credibility** The main focus in revising the Graph Coloring-Based algorithm was to improve the level of credibility. The original algorithm demonstrated a very low level of credibility despite the use of a blind recognizer which extracts the watermark value. By making minor changes to the algorithm the Graph Coloring-Based$_{SM}$ algorithm demonstrates a very high level of credibility. It is guaranteed to recognize the watermark in a watermarked, but otherwise unaltered program. There is also a very low probability that a watermark recovered was recovered by coincidence due to the tight constraints applied to choosing the triples during embedding and recognition. This is a definite improvement over the original Graph Coloring-Based algorithm which left loopholes in the proof of ownership.

**Data-rate** In the original algorithm the upper bound for the data-rate was the number of vertices in the interference graphs. In order to improve the credibil-
ity we further constrained the number of vertices which could be used during the embedding process. This has the effect of further decreasing the data-rate for the Graph Coloring-Based\textsubscript{SM} algorithm.

To examine the data-rate of the Graph Coloring-Based\textsubscript{SM} algorithm we watermarked the ten test applications. For each application we were unable to embed even a single byte watermark. One reason for this failure is that during the embedding process the watermark is converted to a binary string and an 8-bit length field is prepended. The length field represents the length of the binary string and is used during recognition. Unfortunately, this necessitates us embedding an additional byte for the single byte watermark. Additionally, we discovered that the more complex the method the fewer bits we are able to embed. This is because the complex methods also have very complex graphs with many interferences. This drastically decreases the chances of obtaining three non-interfering nodes, let alone several of these triples.

In our previous evaluation we attempted to increase the data-rate by performing static method inlining. After the inlining only one of the ten applications could hold 16 or more bits. To try and further improve the data-rate in this evaluation we have also performed dynamic method inlining. The data-rate obtained both with and without inlining for the current set of test applications can be seen in Table 8.1. The static and dynamic inlining did increase the data-rate for seven of the ten test applications. However, even after inlining only two of the applications could hold 16 or more bits, which is necessary to embed a single byte watermark.

**Stealth** As with the original algorithm, a watermark embedding using the Graph Coloring-Based\textsubscript{SM} algorithm is extremely hard to detect. In the embedding process no additional code is added to the application. Instead, the `load` and `store` instructions are altered so that they reference different local variable numbers. To discover that the local variable assignment was artificially imposed would require extensive analysis.

In an attack against the Graph Coloring-Based\textsubscript{SM} algorithm an adversary may
<table>
<thead>
<tr>
<th>Application</th>
<th>Without inlining</th>
<th>With inlining</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fit</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>illness</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>lu</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>machineSim</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>matrix</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>probe</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>puzzle</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>CaffeineMark</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Java Grande</td>
<td>12</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 8.1: Bits embedded using the Graph Coloring-Based$_{SM}$ algorithm both with and without inlining.

examine the static statistics of the application. Since the algorithm only modifies the local variable assignment, the static statistics remain unchanged when an application is watermarked with the exception of the scalars. The scalars statistic measures non-array variables and this includes the local variables. A decrease in the scalar variables is noticed because the local variable assignment used by the algorithm is often more efficient than that found in Java applications compiled using the javac or jikes compilers. This change in statistics could cause an adversary to perform a more detailed manual study, but we believe the change in local variable assignment will be hard to detect. Based on this we conclude that the Graph Coloring-Based$_{SM}$ algorithm demonstrates a high degree of stealth.

**Overhead** The Graph Coloring-Based$_{SM}$ algorithm introduces no additional code and makes only minimal code alterations. Because of this the impact on performance and size is negligible. However, if inlining is required to increase the data-rate then an increase in application size will be noticed. For the two applications in which inlining was useful, illness suffered a 59% increase in size while the increase for Java Grande was more modest at 7%.
Robustness  Just like the original Graph Coloring-Based algorithm the Graph Coloring-Based$_{SM}$ is susceptible to a variety of attacks. In fact, it is vulnerable to all of the same attacks. The algorithm is still vulnerable to any attack which is likely to reassign the registers such as a decompile, recompile attack or register reallocation. Of the 35 SANDMARK obfuscations, the Graph Coloring-Based$_{SM}$ is also theoretically vulnerable to 28. Those obfuscations which will leave the watermark intact include branch inverter, constant pool reoderer, false refactor, field assignment, objectify, overload names, and publicizer.

Now that the credibility of the algorithm has improved we are able to perform a partial evaluation with respect to strength against the various obfuscations and the potential performance implications associated with destroying the watermark. Due to the low data-rate we are only able to test the practical strength against distortive attacks using the inlined versions of illness and Java Grande. Of the 35 obfuscations 14 were able to destroy the embedded watermark in at least one of the two applications. Results from applying the obfuscations to the two test applications are shown in Table 8.2.

For each obfuscation which destroyed the watermark embedded in Java Grande we evaluated the effects the obfuscation had on performance. For this particular test application 13 obfuscations destroyed the watermark. The effect on performance was quite minimal for most of the obfuscations. For example, irreducibility and variable reassigner destroyed the watermark with no impact on performance while class splitter and static method bodies had an impact of only 1%. Of the obfuscations only two had a significant impact: promote primitive types (267%) and exception branches (6640%). From this we can see that it is easy to destroy the watermark with little impact on the application.

8.2 Arboit Algorithm

One of the yet unanswered questions in the area of software watermarking is whether dynamic algorithms are inherently more resilient to attacks than static algorithms.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array folder</td>
<td>⊕</td>
</tr>
<tr>
<td>Array splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Block marker</td>
<td>⊕</td>
</tr>
<tr>
<td>Bludgeon signatures</td>
<td>⊕</td>
</tr>
<tr>
<td>Boolean splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Branch inverter</td>
<td>⊕</td>
</tr>
<tr>
<td>Buggy code</td>
<td>⊕</td>
</tr>
<tr>
<td>Class splitter</td>
<td>⊕</td>
</tr>
<tr>
<td>Constant pool reoderer</td>
<td>⊕</td>
</tr>
<tr>
<td>Dublicate registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Dynamic inliner</td>
<td>⊕</td>
</tr>
<tr>
<td>Exception branches</td>
<td>⊕</td>
</tr>
<tr>
<td>False refactor</td>
<td>⊕</td>
</tr>
<tr>
<td>Field assignment</td>
<td>⊕</td>
</tr>
<tr>
<td>Inliner</td>
<td>⊕</td>
</tr>
<tr>
<td>Insert opaque predicates</td>
<td>⊕</td>
</tr>
<tr>
<td>Interleave methods</td>
<td>⊕</td>
</tr>
<tr>
<td>Irreducibility</td>
<td>⊕</td>
</tr>
<tr>
<td>Merge local integers</td>
<td>⊕</td>
</tr>
<tr>
<td>Method merger</td>
<td>⊕</td>
</tr>
<tr>
<td>Objectify</td>
<td>⊕</td>
</tr>
<tr>
<td>Opaque branch insertion</td>
<td>⊕</td>
</tr>
<tr>
<td>Overload names</td>
<td>⊕</td>
</tr>
<tr>
<td>Parameter aliases</td>
<td>⊕</td>
</tr>
<tr>
<td>Promote primitive registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Promote primitive types</td>
<td>⊕</td>
</tr>
<tr>
<td>Publicizer</td>
<td>⊕</td>
</tr>
<tr>
<td>Random dead code</td>
<td>⊕</td>
</tr>
<tr>
<td>Rename registers</td>
<td>⊕</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>⊕</td>
</tr>
<tr>
<td>Reorder parameters</td>
<td>⊕</td>
</tr>
<tr>
<td>Split classes</td>
<td>⊕</td>
</tr>
<tr>
<td>Static method bodies</td>
<td>⊕</td>
</tr>
<tr>
<td>String encoder</td>
<td>⊕</td>
</tr>
<tr>
<td>Variable reassigner</td>
<td>⊕</td>
</tr>
</tbody>
</table>

⊕ : watermark found in all test applications
⊕ : watermark destroyed in at least one test application

Table 8.2: Results of applying the SANDMARK obfuscations on two of the test applications which were watermarking using the Graph Coloring-Based$_{SM}$ algorithm.
One technique to investigate this idea is to develop, implement, and evaluate a dynamic version of an already known static algorithm. To this end we have developed and implemented dynamic versions of the two algorithms proposed by Arboit [18], A1 and A2 (DA1 and DA2 respectively).

DA1 and DA2 are execution trace watermarking algorithms because the watermark is embedded in the trace of the program as it is run with a specific input. The novel aspect of DA1 and DA2 is that the execution trace is used to identify the set of program branching points $B$ instead of using randomly selected points. The motivating factor in this design is that the execution of the program will execute the original set of branching points when run with the secret key no matter how distorted an attacker makes the application. This assumption is based on the idea that most transformations that cause the execution to skip the branch will most likely alter the functionality of the application. Thus the dynamic nature will improve the algorithm’s ability to withstand distortive attacks.

The set $B$ of program branching points is required for both the embedding and recognition phases. $B$ is constructed by annotating the application prior to execution. Figures 8.4(a) and 8.4(b) illustrate the transformation that occurs due to the annotations. To illustrate the embedding procedure, suppose the execution of the application using the secret input takes the path $\{1, 2, 4, 9, 10, 11, 13, 14\}$. Thus the set $B$ consists of the `if` instructions in blocks 2 and 10. To watermark this method either A1 or A2 is used. In this example, the transformation that occurs due to watermarking is illustrated in Figure 8.4(c).

The recognition set $B$ is again acquired through annotating the watermark application and an execution trace. To continue with the example, the execution trace consists of the blocks $\{1, 2, 4, 9, 10, 10', 11, 13, 14\}$. What we see is that the opaque predicate inserted in block 2’ is not executed. This is because Java uses short circuit evaluation so the second predicate does not necessarily need to be evaluated. (In the current implementation all inserted predicates are opaquely true.) Since the trace identified block 2 we can still recover the opaque predicate in 2’. This is accomplished by examining the fall through block of every `if` instruction identified
Figure 8.4: The watermarking of a method using DA1 or DA2 requires an annotation phase which allows us to identify which branch instructions are executed in the trace.
in the trace since it is a possible opaque predicate.

8.2.1 Evaluation

Other than making use of an execution trace to select branching points, DA1 and DA2 are identical to the static algorithms A1 and A2. Because of this, the evaluation results with respect to the properties of credibility, data-rate, stealth, and overhead are the same as the static evaluation results. In converting the algorithm from static to dynamic, we were very interested in seeing if such a change would have an impact on the robustness of the algorithm.

Robustness The Dynamic Arboit algorithms still embed the watermark through the addition of carefully selected opaque predicates. The dynamic nature of the algorithm does not improve the resiliency against subtractive attacks. If the attacker is familiar with number theory he may realize that the watermarked application contains opaque predicates. Once the opaque predicates are removed the protection has been subverted. Additionally, it is still crucial to maintain the secrecy of the opaque predicate library.

The Dynamic Arboit algorithms are also still susceptible to a variety of automated distortive attacks. In fact, they are theoretically still vulnerable to the same 18 SandMark obfuscations to which the static versions are vulnerable. Any obfuscation which uses opaque predicates has the potential to disrupt watermark recovery if an opaque predicate used in the obfuscation also appears in the opaque predicate library used by watermarking algorithm. In SandMark these obfuscations include block marker, buggy code, interleave methods, irreducibility, insert opaque predicates, and opaque branch insertion. Because recognition is ultimately based on a textual comparison between a predicate identified in the application and the predicates in the opaque predicate library, any obfuscation which alters the instruction sequence will prevent accurate recovery. For example, promoting a primitive type to its corresponding object type, e.g. int to java.lang.Integer. Additionally, DA2 is susceptible to techniques like method inlining and method merging where
the opaque method is no longer identifiable.

In practice we found that the DA1 algorithm is only slightly more resistant to distortive attacks than the static version, A1. Of the 35 SANDMARK obfuscations seven were able to destroy a watermark embedded using DA1. This is a decrease of two compared to the static version. For DA2 the resistance to distortive attacks was the same as the static version; ten of the 35 SANDMARK obfuscations were able to destroy the watermark. The results from applying the obfuscations are shown in Table 8.3.

Both DA1 and DA2 are still susceptible to the same additive attacks as the static Arboit algorithms. More specifically, embedding an additional watermark using either DA1 or DA2, regardless of which was used for the original embedding, has the effect of destroying the original as well as the new watermark. Additionally, of the 11 watermarking algorithms presented in the previous chapter, only Graph Coloring-Based and Abstract Interpretation could not be used to launch an additive attack against the Dynamic Arboit techniques.

8.3 Discussion

In this chapter we presented results from our in-depth study of the Graph Coloring-Based and Arboit algorithms. For both of these algorithms our goal was to improve overall strength while maintaining the essence of the original idea. What we found from these two studies is that reworking a technique may not be the best way to develop a stronger watermarking algorithm.

In the case of the Graph Coloring-Based algorithm we focused on improving recognition reliability. To do so required us to place additional constraints on the embedding process. This lead to a significant decrease in the data-rate. Additionally, through the improved implementation we were able to show that the Graph Coloring-Based algorithm is in fact vulnerable to a variety of simple distortive attacks. Overall, our conclusion is that the Graph Coloring-Based algorithm is not really suitable for watermarking software.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>DA1</th>
<th>DA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array folder</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Array splitter</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Block marker</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Bludgeon signatures</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Boolean splitter</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Branch inverter</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Buggy code</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Class splitter</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Constant pool reorderer</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Dublicate registers</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Dynamic inliner</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Exception branches</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>False refactor</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Field assignment</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Inliner</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Insert opaque predicates</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Interleave methods</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Irreducibility</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Merge local integers</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Method merger</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Objectify</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Opaque branch insertion</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Overload names</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Parameter aliases</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Promote primitive registers</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Promote primitive types</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Publicizer</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Random dead code</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Rename registers</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Reorder instructions</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Reorder parameters</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Split classes</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Static method bodies</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>String encoder</td>
<td>✫</td>
<td>✫</td>
</tr>
<tr>
<td>Variable reassigner</td>
<td>✫</td>
<td>✫</td>
</tr>
</tbody>
</table>

⊕ : watermark found in all test applications
⊕ : watermark destroyed in at least one test application

Table 8.3: Results of applying the SandMark obfuscations on nine of the test applications which were watermarked using the Dynamic Arboit algorithms.
For the Arboí algorithms we presented a novel extension to study static versus dynamic watermarking algorithms. We wanted to know if the robustness of an algorithm could be improved by converting it from static to dynamic. Through evaluation we showed that DA1 was only minimally stronger than A1 and that there was no difference between DA2 and A2. From this we conclude that it is not clear that converting a known static algorithm will improve the strength. However, this does not indicate that the class of dynamic algorithms is not inherently stronger.
CHAPTER 9

BRANCH-BASED SOFTWARE WATERMARKING

In Chapter 7 we presented the details of 11 published software watermarking algorithms. Through a thorough evaluation of these algorithms we highlighted both the strengths and weaknesses. From this evaluation we identified a set of shortcomings from which the majority of the watermarking techniques suffer. In Chapter 8 we presented the results from our in-depth study of two software watermarking algorithms. In this work we attempted to improve on the algorithms by making modifications which maintain the essence of the original techniques. In the Graph Coloring-Based algorithm we focused on improving recognition reliability, but this came at the cost of the data-rate. With the Arboit algorithm we attempted to improve the resilience against attack by converting it from static to dynamic. From these in-depth studies we realized that instead of trying to rework the techniques, the best way to address the set of deficiencies was to develop a completely new watermarking scheme.

In this chapter we present a novel dynamic software watermarking algorithm which was designed to improve on the common vulnerabilities of previous algorithms. The algorithm incorporates ideas from code obfuscation (to aid in preventing reverse engineering) and software tamper detection (to thwart attacks such as the application of semantics-preserving transformations). The main idea is to incorporate a specifically designed branch function (see Section 3.5) which will generate the fingerprint as the program executes. This basic idea can be implemented in a variety of different ways. Many of the implementation details are architecture specific; however, variations can be constructed based on the architectural specifications. In this chapter we describe the technique and how it can be implemented to watermark x86 native executables and Java applications.
9.1 The Branch-Based Algorithm

The Branch-Based software watermarking algorithm is a dynamic fingerprinting algorithm. Instead of directly embedding the fingerprint in the code, code is embedded which generates the fingerprint as the program executes. Because the algorithm is dynamic, a secret key $\mathcal{K}$ is necessary for both embedding and recognition. $\mathcal{K}$ is a sequence of inputs to the program $I_0, I_1, \ldots$ which will force a particular execution of the program.

A general overview of the embedding and recognition process is illustrated in Figure 9.1. The embedding process begins with a program $P$, an authorship mark $AM$, a secret seed value, and a secret key $\mathcal{K} = I_0, I_1, \ldots$. Central to the design of the Branch-Based algorithm is a specially designed branch function. (For the definition of a branch function see Section 3.5.) If constructed properly, the branch function makes it possible to simultaneously embed an authorship and a fingerprint mark using a single watermarking algorithm.

At ($\textcircled{a}$), $P$ is modified to embed the authorship and fingerprint marks. In the first step of this process, the program is executed using the secret input sequence to identify an execution path. The fingerprint mark is generated through repeated execution of the branch function. To incorporate the branch function a set of branch instructions are selected which reside on the execution path associated with the secret input sequence $I_0, I_1, \ldots$. The selected branch instructions are replaced with a function call to the inserted branch function. Unlike the fingerprint mark, the authorship mark is directly embedded in the code of the branch function. The result of this process is a new program $P'$.

As the program executes, the branch function evolves a key which is seeded with a secret seed value. To extract the fingerprint mark the program is executed at ($\textcircled{b}$) using the secret input sequence $I_0, I_1, \ldots$. The final key generated is extracted from the key storage at ($\textcircled{c}$), which could be a runtime variable, to yield the fingerprint.

This type of embedding and recognition process deviates from the standard definition of a software watermarking system. To simultaneously embed an authorship
and fingerprint mark the \textbf{embed} function should have four inputs and two outputs.

\[ \text{embed}(P, AM, \mathcal{K}, \text{seed}) \rightarrow P', FM \]

By definition, the authorship mark \( AM \) should be the same for every copy of the program. Similarly, \( \mathcal{K} \), which is linked to \( AM \) should be the same for every copy. A fingerprint mark for a particular instance of a program is based on the program execution and the \textit{seed}. Thus, the \textit{seed} must be unique for each copy. Additionally, this means the fingerprint mark \( FM \) is generated during embedding and is an output of the \textbf{embed} function.

Similarly, the \textbf{recognize} function is non-standard with two inputs and two outputs.

\[ \text{recognize}(P', \mathcal{K}) \rightarrow AM, FM \]

The recognition technique is blind thus the authorship and fingerprint marks can
be extracted from the program by providing the secret input.

Figure 9.2 illustrates a possible transformation based on the Branch-Based algorithm using a simple C program. In this simple example, when the watermarked program is executed, the global variable key is evolved. The final value stored in the variable key represents the fingerprint for the program. By analyzing the program in this example it can be seen that even if a different input value was used the key evolution will remain the same. This will be true for most small programs. To create variation in the key evolution for different input values, we would also have had to change the function call factorial(x-1) to the call branchFunction(x-1). As we elaborate on the algorithmic details the reason the function call was not modified will become clear.

9.2 Fingerprint Branch Function

The branch function was originally created as an obfuscation technique to disrupt static disassembly by exploiting the assumption that a function call returns to the instruction immediately following the call instruction [64]. To accomplish this, execution is routed through the branch function. Through a series of calculations which were described in Section 3.5, the branch function determines the location of the next instruction. On completion of the branch function, execution resumes at the identified instruction. In the Branch-Based watermarking algorithm we extend the functionality of the branch function to create what we call a fingerprint branch function (FBF).

The design of the Branch-Based algorithm revolves around the repeated execution of the fingerprint branch function to generate the fingerprint mark for the program. Figure 9.3 illustrates a single execution of the FBF which progresses through three steps. The first step is to perform an integrity check of the program producing some value $v_i$. An integrity check is an inserted section of code used to verify the integrity of the program and to identify active debugging [80]. For example, one of the integrity checks could choose a block of code and calculate its checksum. If the
```c
void main(int argc, char *argv[]){
    int x = atoi(argv[1]);
    printf("%d! = %d\n", x, factorial(x));
}

int factorial(int x){
    if(x == 1)
        return x;
    return (x * factorial(x-1));
}
```

```c
long key = seed;

void main(int argc, char *argv[]){
    int x = branchFunction(argv[1]);
    branchFunction("%d! = %d\n", x, branchFunction(x));
}

int factorial(int x){
    if(x == 1)
        return x;
    return (x * factorial(x-1));
}

void branchFunction(void *x){
    key = evolveKey(key);
    return;
}
```

Figure 9.2: Example of a simple C program before and after being watermarked with the Branch-Based algorithm.
attacker attempts to store break points or apply a semantics-preserving transformation, even if the modification is very slight, the checksum will be incorrect. When trying to detect the presence of a debugger, the elapsed time of execution from one point to another can be used as an integrity check. Through the incorporation of an integrity check the FBF can provide tamper detection throughout the entire application.

In step two, the integrity check value $v_i$ and the current key $k_i$ are used as inputs to generate the next key $k_{i+1}$. Key generation within the FBF is accomplished through the use of a secure one-way hash function such as SHA [89]. The idea behind the use of a one-way function is that it is easy to compute $y = f(x)$, but given $y$, it is difficult to reverse the process and find $x$. If an attacker executes the program and extracts the final key generated, he will not be able to use that information to calculate the sequence of keys generated. This is an important feature in preventing attacks based on static analysis of the program.

The key generated in step two is then used in the final step of the FBF. There are a variety of ways this particular step of the FBF can be implemented and many of the details are architecture specific. The general idea is to use the key $k_{i+1}$ to identify the instruction where execution will resume upon completion of the FBF. This particular instruction is the target of the replaced branch instruction. Specific techniques for implementing this step will be presented for native executables and
Figure 9.4: Illustration of a single step in the key evolution process.

Java applications.

It is important to note that while the integrity check value is calculated in step one, the actual tamper detection does not occur until step three. Additionally, the earliest point of program failure occurs after the FBF has completed execution. Instead of comparing $v_i$ with an expected value, it is used in controlling the execution of the program via the key generation. This is considered a stronger form of tamper detection since it is often easy for an attacker to remove the comparison.

A final important aspect of the FBF is that it can be used as the embedding location for an authorship mark if one is desired. An ideal authorship mark will possess some mathematical property that allows for a strong argument that it was intentionally placed in the program and that its discovery is not accidental. For example $AM = pq$ where $p$ and $q$ are very large primes would be a good candidate. Because factoring is a hard problem only the person who embedded such a water-mark would be able to identify the factors $p$ and $q$. To encode such an authorship mark in the FBF, a secure one-way hash function is chosen such that one of the variables can be set to $AM$. A possible example is

$$k_{i+1} = \text{SHA1}[(k_i \oplus AM) \parallel v_i]$$

In this example, we xor the current key with the authorship mark and then we concatenate the integrity check value. Figure 9.4 illustrates a single step in the key evolution process when the authorship mark is incorporated.

In summary, the FBF performs the following tasks:
Figure 9.5: Overview of how the **Branch-Based** algorithm watermarks an application.

- An integrity check which produces the value \( v_i \).
- Generation of the next function key, \( k_{i+1} \), through the use of a one-way function, the integrity check value, and the previous key, \( k_{i+1} = g(k_i, v_i) \).
- Transfer of execution to the instruction which was the target of the replaced branch instruction using \( k_{i+1} \).

### 9.3 The Branch-Based Algorithm – Native Code Implementation

The **Branch-Based** watermarking algorithm is a dynamic fingerprinting algorithm in which the fingerprint is generated as the program executes. To make the algorithm as strong as possible, the fingerprint generating mechanism draws on features specific to the program’s execution architecture. In our native code implementation, the watermark embedding and extraction process is performed in several steps (see Figure 9.5).
Annotation Prior to embedding the authorship mark and fingerprint generating code, annotation points must be added throughout the program. These points are in the form of a function call, e.g. `mark()`, and can be inserted by the user at development time or through an automated process as part of the watermarking procedure. The `mark()` calls indicate sections of code which can be safely modified to aid in fingerprint generation.

Tracing The fingerprint of the program is linked to the execution path exercised by the secret input sequence. After the program has been annotated, a tracing run of the program is performed using the secret input sequence. As the program executes annotation points are encountered. These points will be used to embed the fingerprint generating code.

Embedding During the embedding phase the `mark()` calls are removed. Within a section of code associated with an encountered `mark()` call, branch instructions are selected. These branch instructions are replaced with calls to the fingerprint generating mechanism, the FBF. Finally, the FBF is constructed such that it encodes the authorship mark after which it is inserted in the program.

Extraction To extract the fingerprint, the program is executed using the secret input sequence. This will force the program to exercise the same execution path as when it was run during the tracing phase. Once the program has finished executing the input sequence, the final key generated is extracted from the key storage. This key represents the fingerprint for the program.

In the following sections we consider each of the steps in the native code implementation in detail.

Our implementation of the Branch-Based watermarking algorithm for native code is accomplished by disassembling a statically linked binary, modifying the instructions, and then rewriting the instructions to a new executable file. The current prototype is designed to watermark Windows executable files.
9.3.1 Annotation

The annotation phase can either be performed manually by the user during the development of the program or as an automated procedure during the watermarking process. In either case, during annotation a call to the function `mark()` is inserted at the beginning of each function in the program which can safely be modified to aid in fingerprint generation. When the annotation phase is applied at the source code level to the simple C program in Figure 9.2 the transformation yields the following:

```c
void main(int argc, char *argv[]){
   int x;
   mark();
   x = atoi(argv[1]);
   printf("%d! = %d\n", x, factorial(x));
}

int factorial(int x){
   mark();
   if(x == 1)
      return x;
   return (x * factorial(x-1));
}
```

In our implementation of the Branch-Based algorithm the annotation phase is automated and performed at the assembly code level. During program linking a map file can be generated. This file contains the name of each function in the program along with its start address. Using this information we insert a single call instruction at the beginning of each function in the program. Figure 9.6 illustrates this transformation. Note that the call instruction is the very first assembly instruction. That is, it was inserted prior to establishing the stack frame for the function. This will be a useful detail during the tracing phase.

9.3.2 Tracing

The goal of the tracing phase is to identify the set of functions which are executed as the program is run with the secret input sequence. This is accomplished through the functionality of the `mark()` function. Each time the `mark()` function is executed
Figure 9.6: Transformation resulting from the automated annotation phase.
the return address is obtained from the stack. The value of the return address is the address of the instruction proceeding the call mark instruction in the calling function, i.e. the first true instruction in the calling function (push ebp in Figure 9.6 (b)). By subtracting the length of the call mark instruction (5 bytes) from the return address, we have identified the start of the function. This value is recorded in the trace file. At the end of the tracing run we have gathered a list of start addresses for each function executed during the trace. The trace will also indicate if a function has been executed more than once.

9.3.3 Embedding

Once a trace of the program has been completed the authorship mark and fingerprint generating code can be embedded. To carry out this phase of the watermarking process four inputs are required: the execution trace, the authorship mark, the fingerprint seed, and the program. Using these inputs the embedding process is broken into five steps:

1. The fingerprint branch function is constructed and added to the program.

2. From each function identified in the trace, a set of branch instructions are selected.

3. The selected branch instructions are replaced with a call to the fingerprint branch function.

4. A table is constructed and inserted in the data section of the program.

5. The fingerprint mark for the program is calculated.

As a result of this process a new Windows executable is produced which when executed with the special input sequence will generate the fingerprint in the key storage. An additional output of this process is the fingerprint mark assigned to the particular program instance.
Fingerprint Branch Function Construction

In the native code version of the FBF the fingerprint generation is linked to program execution by using the key to identify the correct displacement to the branch target. In this particular implementation the branch-target displacements associated with each replaced branch are stored in a table. The key is used to access the table. The displacement obtained is added to the return address that was stored on the stack when the FBF was called. After the execution of the FBF is completed, the next instruction to execute is the target of the original branch instruction.

The first aspect in the construction of the FBF is to select an integrity check procedure. For this implementation we used a single simple checksum integrity check. When triggered the integrity check will calculate a checksum over a block of instruction bytes. The instruction bytes are chosen by seeding a random number generator with the current key. Using the random number generator we select the location of the start byte as well as the number of consecutive bytes to be checked such that the number of bytes checked is in the range [0, 500].

It is important to note that this is a very simple integrity check that is only used to illustrate the capabilities of the Branch-Based watermarking algorithm. In practice a variety of stealthy integrity checks would be used. Ideally the checks would be customized to address the specific requirements of the application. One of the limitation of the use of integrity checks is that describing them in detail can decrease their potency. This is often true for techniques aimed at providing software tamper resistance.

The second task for the FBF is to calculate the next key based on the current key \( k_i \) and the integrity check value \( v_i \). This aspect of the FBF is constructed as previously described. \( k_{i+1} \) is generated using the secure hash function SHA-1 with an input constructed from \( k_i, v_i \), and the authorship mark:

\[
    k_{i+1} = \text{SHA1}[(k_i \oplus AM) \parallel v_i]
\]

In the current implementation of the Branch-Based algorithm each time a new
key is generated it is stored in the currentKey global variable. This is an obvious limitation because once the attacker has identified the variable, monitoring it is an easy task. Such monitoring could be made more difficult through the selection of integrity checks. An alternate solution could be to leverage secure computing devices such as the Trusted Platform Module (TPM) which is available in all of IBM’s ThinkPad laptops.

As part of the embedding process a table is constructed and inserted in the data section of the binary. In this table we store the branch-target displacements for each branch replaced. The table is organized such that the branch-target displacement associated with a particular key is stored in the slot which is the hash of the key.

\[ d_{i+1} = T[h(k_{i+1})] \]

The displacement identified is the number of bytes from the byte immediately following the branch instruction (i.e. the start of the next instruction) to the start of the target instruction. Stored in the return address for the FBF is the address of the instruction immediately following the converted branch instruction. Thus, to identify the location of the target instruction we add the displacement \( d_i \) to the return address. Similar to the original branch function, we exploit the ease with which the value of the return address on the stack can be modified so that the FBF returns to the target instruction. Figure 9.7 provides an example set of assembly instruction in which the displacement is obtained and the return address is modified.

In summary, the native code version of the FBF is constructed such that it performs the following tasks:

- A simple checksum integrity check producing the value \( v_i \).
- Generation of the next key \( k_{i+1} \) using the secure hash function SHA-1, \( k_i \), \( v_i \), and \( AM \).
- Identification of the displacement to the target instruction via \( d_{i+1} = T[h(k_{i+1})] \), where \( T \) is the table in the data section and \( h \) is a hash function.
mov ecx, ebp+4  ; put return address in ecx
mov eax, T[eax+4]
sar eax, 1
jnb L1
add ecx, eax
mov esp, ebp
pop ebp
pop eax        ; pop old return address
add esp, 4
push eax       ; push old return address
push ecx       ; push new return address
jmp L2

<table>
<thead>
<tr>
<th>L1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>add ecx, eax</td>
</tr>
<tr>
<td>mov ebp+4, ecx</td>
</tr>
<tr>
<td>mov esp, ebp</td>
</tr>
<tr>
<td>pop ebp</td>
</tr>
<tr>
<td>pop eax</td>
</tr>
<tr>
<td>add esp, 4</td>
</tr>
<tr>
<td>push eax</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

Figure 9.7: Example assembly instructions used in the FBF to modify the return address.

- Computation of the return location by adding the displacement $d_{i+1}$ to the return address.

Once the FBF and its support functionality (such as the integrity check code) have been constructed, they are added to the program.

### Branch Instruction Selection

During the tracing phase the program was executed using the secret input sequence to obtain a trace of the program. The trace identified a set of functions $F$ through which execution passes. In each function $f \in F$ the set of branch instructions $B_f = b_1, \ldots, b_k$ to be converted is selected. Special care must be taken in selecting which branches are converted. The branch instructions used in any given function must reside on a path that will be traversed every time the function executes. Without imposing this constraint an irregular key evolution will occur which will result in transferring execution to an incorrect instruction. This will ultimately lead to
improper program behavior.

Figure 9.8 illustrates the irregular key evolution which will occur if branches are selected along a non-deterministic path. Suppose we select the jge in block 0, the jmp in block 1, and the call in block 3. When the path 0, 1, 3, 4 through the function is executed we obtain the key evolution associated with \( @ \). However, when the path 0, 2, 3, 4 is executed a different key evolution occurs. When the FBF is called from block 3 the keys calculated from these two executions will differ \( (k_3 \neq k'_3) \). Based on this simple example, maintaining an entry for both keys seems feasible. Unfortunately, as the complexity of the function increases and loops are introduced, such a solution becomes infeasible. Not only would we have to identify all paths through the function, but based on these paths the size of the table would rapidly increase in size.

To eliminate the possibility of an irregular key evolution the set \( B_f \) is comprised of all branches which reside along the deterministic path through the function. In order to identify the deterministic path we compute the dominator set for the exit block in the function control flow graph. The dominator set may include blocks which are part of a non-deterministic path such as a loop header. Any such block is removed from the path.

**Branch Instruction Replacement**

Once we have identified the set of branch instruction \( B_f \) for a function \( f \), each \( b_i \in B_f \) is replaced with a call to the FBF. For each branch replaced a mapping must be constructed between the branch-target displacement and a key \( (k_i \rightarrow d_i) \). The key must be calculated exactly as it would during program execution. To accomplish this the branch instructions are replaced in execution order by using the dominator set for the exit block.

The evolution for each function begins with the initial key \( k_0 \). The value of \( k_0 \) is derived from the seed which was an input to the embedding process. For each function the value of \( k_0 \) must be slightly different so that each branch-target displacement is associated with a unique key. To illustrate, suppose \( k_0 \) is the same
Figure 9.8: Illustration of the irregular key evolution which can occur if branches are selected along a non-deterministic path.

for two function $f_1$ and $f_2$. To have a key evolution where $k_{i+1}^{f_1}$ and $k_{i+1}^{f_2}$ are different would require $v_0^{f_1}$ and $v_0^{f_2}$ to be different. However, the simple checksum integrity check uses the current key value to determine which instruction bytes to check. Thus $k_0^{f_1} = k_0^{f_2} \Rightarrow v_0^{f_1} = v_0^{f_2} \Rightarrow k_{i+1}^{f_1} = k_{i+1}^{f_2}$. To derive $k_0^{f_i}$ we xor the address of the initial instruction in the function $f$ with the seed.

As we replace the branches in execution order the key is evolved following the first two steps of the FBF:

- We calculate the value $v_i$ by performing the simple checksum integrity check.
- We generate the next key $k_{i+1}$ using the secure hash function SHA-1, $k_i$, $v_i$, and $AM$ ($k_{i+1} = \text{SHA1}[(k_i \oplus AM) \parallel v_i]$).
For each \( f \in F \) a mapping is maintained between the calculated keys and the branch-target displacements:

\[
\theta_f = \{ k_f^1 \rightarrow d_f^1, k_f^2 \rightarrow d_f^2, \ldots, k_f^n \rightarrow d_f^n \} \]

**Data Table Construction**

Fingerprint generation and proper program execution are linked by using the current key to identify the branch-target displacement. Each of the branch-target displacements are stored in a table inserted in the data section of the binary. In this step of the embedding process each of the key-displacement mappings \( \theta_f \) are used to construct a single table \( T \). The first step in laying out the table is to select a perfect hash function such that each key maps to a unique slot in the table [45, 67]. It is best to use a minimal perfect hash function so that the table size is minimized.

\[
h = \{ k_1, k_2, \ldots, k_m \} \rightarrow \{ 1, 2, \ldots, n \}, m \leq n
\]

However, in the current implementation we do not use a minimal perfect hash function. The displacements are stored in the table such that \( T[h(k_i)] = d_i \).

**Fingerprint Mark Calculation**

Unlike the authorship mark, the fingerprint mark is not embedded in the program. Instead it is generated as the program is executed. Each function \( f \in F \) obtained by executing the program with the secret input will produce a final function key \( k_f \). Each of these keys is combined in a commutative way, e.g. add the values, to produce the fingerprint mark for the program. The variation in the fingerprint mark is obtained through the seed, which is unique for each copy of the program.

To illustrate, suppose we have a program containing 10 functions \( f_1, f_2, \ldots, f_{10} \). Using the secret input sequence \( I_0, I_1, I_2 \) we obtain the execution trace \( f_1, f_2, f_5, f_2, f_6, f_2, f_{10} \). Based on this execution trace, Figure 9.9 illustrates the fingerprint generation process as the program executes. The fingerprint generation
begins with $FM_0$ which is the seed. As each function in the trace is executed the final key produced $k_f$ is added to fingerprint value $FM_k$. The final fingerprint value, in this case $FM_7$ is the fingerprint.

### 9.3.4 Extraction

As with embedding, the first step in extracting the fingerprint mark is to execute the program using the secret input sequence. This will cause the fingerprint mark to be built in the fingerprint storage. To extract the fingerprint we access the location where the fingerprint is stored. To extract the authorship mark we simply use static analysis to identify the FBF. From here we are able to isolate the one-way function to extract the mark.

### 9.3.5 Registration-Based Customization

The only static variation in differently fingerprinted instances of a program is in the inserted displacement table. This feature enables software companies to produce and distribute fingerprinted software in the traditional manner. The program purchased would be non-functional until the user installs the software and registers it with the company. Upon registration, the fingerprint seed and table are distributed, creating a fully functioning program. Previously, if a software company wanted to tie a specific fingerprint mark to a purchaser, the user had to purchase the software directly from the company and the program was fingerprinted at that point. By using the Branch-Based watermark, distribution of fingerprinted software can be accomplished through prepackaged software sold at retail stores.

By using this model, software can either be distributed as prepackaged software purchased at a retail store or through online download distribution. In either case, the software package contains the installation executables as well as a crippled executable, which is non-functional until the installation process has been completed. During installation, a one-time connection to the registration server is required. The registration process requires that a user submit some form of unique identifica-
Figure 9.9: Illustration of the fingerprint generation as the program executes the secret input sequence $I_0, I_1, I_2$ associated with the execution trace $f_1, f_2, f_5, f_2, f_6, f_2, f_{10}$. 
tion in exchange for the watermark-specific code, which creates the fully executable software linked to the user. The unique identifier must enable the future identification of the user. Additionally, this information has to be personal enough that the user is unlikely to share it with others. An example of such information includes a credit card number. Figure 9.10 illustrates the installation process of watermarked software that is linked to the user.

One important distinction to make between the Branch-Based software watermarking technique and fingerprinting techniques used for media is that the technique is not based on signal processing. A media fingerprint is often embedded by the media player. This makes the technique vulnerable to an attack in which the media player is prevented from actually embedding the mark. In the event of such an attack, the non-fingerprinted media is still playable. When a piece of software is prepared for fingerprinting using the Branch-Based technique, the proper control flow is removed. The control flow is added back into the program when the fingerprint is embedded because the execution behavior is tied to the generation of the fingerprint. If an attacker blocks the embedding of the fingerprint, the program is non-functional.
9.3.6 Evaluation

In this section we provide a complete evaluation of the native code implementation of the Branch-Based algorithm. To perform this evaluation we created a prototype implementation for watermarking Windows executable files. The evaluation was conducted using the threat model and the five evaluation properties described in Chapter 5. All experiments were run on a 1.8 GHz Pentium 4 system with 512 MB of main memory running Windows XP Professional. To evaluate the overall strength of the implementation we used the 12 applications in the SPECint-2000 benchmark suite. The Benchmark applications were compiled using Microsoft’s Visual Studio C++ 6.0 with optimizations disabled. The configuration file used to both build and run the benchmark applications can be found in Appendix C. Limitations of our disassembly and rewriting tools required the use of the flag -D Console which yields well formed functions, but decreases the number of branch instructions. Additionally, the limitations prohibited us from analyzing and watermarkings some of the functions in the applications.

Credibility

The Branch-Based algorithm uses a blind recognizer to extract the authorship and fingerprint marks. This type of recognizer provides a high level of credibility because it is guaranteed to recognize the marks in a watermarked but otherwise unaltered program.

Data-Rate

To embed the authorship mark, the Branch-Based algorithm adds code to the program. This type of embedding technique enables any size authorship mark to be embedded in any program.

The size of the fingerprint mark is influenced by two factors. The first is the choice of secure one-way function used in the fingerprint branch function. SHA-1 operates on an input which is less than $2^{64}$ bits in length and produces a 160-bit
value. Thus, each key produced will be 160-bits regardless of the size of the program. More importantly, each function key will be 160-bits. The second factor which will influence the data-rate is the number of methods in the trace. The greater the number of entries in the trace the larger the fingerprint mark. Overall, regardless of the size of the program the fingerprint mark will be at least 160 bits and will increase as the size of the trace increases.

Stealth

Overall, the Branch-Based algorithm demonstrates a relatively low level of stealth. The major factor which compromises the stealth of the algorithm is the number of instructions throughout the watermarked program which call the same function. For many programs this behavior would be highly unusual. However, because the algorithm is dynamic this does not reveal the fingerprint mark until the program is executed.

In the design of the Branch-Based algorithm a different approach was taken with regard to stealth. Instead of trying to create a highly stealthy algorithm which could force us to compromise other aspects of the algorithm, we chose not to hide identifying aspects of the algorithm. Such a design decision puts the authorship mark and fingerprint generating code in plain site. However, we incorporated robustness features which increase the work required by the attacker to remove this code. Thus, the overall strength of the algorithm does not rely on hiding something from the attacker.

Overhead

The overall performance of the watermarked program was evaluated using the SPEC reference inputs. The execution times reported were obtained through five runs. The highest and lowest values were discarded and the average was computed for the remaining three runs.

As can be seen in Table 9.1 very little performance penalty is incurred by the
<table>
<thead>
<tr>
<th>Program</th>
<th>Branches</th>
<th>Execution Time (sec)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original $(T_0)$</td>
<td>Watermarked $(T_1)$</td>
<td>Slowdown $(T_1/T_0)$</td>
</tr>
<tr>
<td>gzip</td>
<td>39</td>
<td>434.24</td>
<td>433.95</td>
<td>-0.67%</td>
</tr>
<tr>
<td>vpr</td>
<td>532</td>
<td>479.12</td>
<td>480.62</td>
<td>0.31%</td>
</tr>
<tr>
<td>gcc</td>
<td>2280</td>
<td>204.07</td>
<td>204.42</td>
<td>0.17%</td>
</tr>
<tr>
<td>mcf</td>
<td>14</td>
<td>552.32</td>
<td>552.77</td>
<td>0.08%</td>
</tr>
<tr>
<td>crafty</td>
<td>118</td>
<td>315.64</td>
<td>1366.77</td>
<td>333.01%</td>
</tr>
<tr>
<td>parser</td>
<td>354</td>
<td>508.93</td>
<td>630.27</td>
<td>23.84%</td>
</tr>
<tr>
<td>eon</td>
<td>1508</td>
<td>993.49</td>
<td>1000.14</td>
<td>0.67%</td>
</tr>
<tr>
<td>perlmonk</td>
<td>627</td>
<td>485.03</td>
<td>482.93</td>
<td>-0.43%</td>
</tr>
<tr>
<td>gap</td>
<td>393</td>
<td>331.89</td>
<td>333.79</td>
<td>0.57%</td>
</tr>
<tr>
<td>vortex</td>
<td>519</td>
<td>323.32</td>
<td>360.22</td>
<td>11.41%</td>
</tr>
<tr>
<td>bzip2</td>
<td>51</td>
<td>840.10</td>
<td>852.22</td>
<td>1.44%</td>
</tr>
<tr>
<td>twolf</td>
<td>341</td>
<td>912.43</td>
<td>922.84</td>
<td>1.14%</td>
</tr>
</tbody>
</table>

Table 9.1: Effect of watermarking on execution time using the native code version of the Branch-Based algorithm.

additional calls and integrity checks for most of the applications. This is because the overhead associated with making the function call is very small. Additionally, the time required to execute the FBF is minimal. Despite this fact, there was a very significant slowdown for the crafty application. We investigated this anomaly and found that some of the watermarked functions were being called repeatedly, thus compounding the impact. To avoid such a large performance impact profiling can be used to identify functions which are not good candidates for watermarking.

The majority of the space increase incurred by the Branch-Based watermark is due to the size of the fingerprint branch function and the displacement table. Since the fingerprint is generated as the program executes the size of the fingerprint does not impact the size of the watermarked program. Additionally, any difference between the converted branch and the call instruction sizes will contribute to the size of the watermarked application. Table 9.2 shows the effect watermarking had on the size of the benchmark applications. For most of the applications the size increase was minimal.

A technique to minimize the size impact is to use a minimal perfect hash function in assigning the slots in the displacement table. Our implementation did not thus the
<table>
<thead>
<tr>
<th>Program</th>
<th>Branches</th>
<th>Table Size</th>
<th>Program Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Original ($S_0$)</td>
</tr>
<tr>
<td>gzip</td>
<td>39</td>
<td>148</td>
<td>102,400</td>
</tr>
<tr>
<td>vpr</td>
<td>532</td>
<td>19,429</td>
<td>208,896</td>
</tr>
<tr>
<td>gcc</td>
<td>2280</td>
<td>218,678</td>
<td>1,646,592</td>
</tr>
<tr>
<td>mcf</td>
<td>14</td>
<td>49</td>
<td>65,536</td>
</tr>
<tr>
<td>crafty</td>
<td>118</td>
<td>1064</td>
<td>303,104</td>
</tr>
<tr>
<td>parser</td>
<td>354</td>
<td>8984</td>
<td>155,648</td>
</tr>
<tr>
<td>eon</td>
<td>1508</td>
<td>117,072</td>
<td>425,984</td>
</tr>
<tr>
<td>perlbmk</td>
<td>627</td>
<td>25,089</td>
<td>737,280</td>
</tr>
<tr>
<td>gap</td>
<td>393</td>
<td>10,627</td>
<td>835,630</td>
</tr>
<tr>
<td>vortex</td>
<td>519</td>
<td>15,369</td>
<td>606,208</td>
</tr>
<tr>
<td>bzip2</td>
<td>51</td>
<td>339</td>
<td>86,016</td>
</tr>
<tr>
<td>twolf</td>
<td>341</td>
<td>8840</td>
<td>311,206</td>
</tr>
</tbody>
</table>

Table 9.2: Effect of watermarking on program size using the native code version of the Branch-Based algorithm.

results could be improved. Table 9.3 illustrates the results which could be expected using a minimal perfect hash function. From this is can be seen that based on an optimal implementation the Branch-Based algorithm has a very minimal impact on the size of the application.

Robustness

Baring the use of a completely secure computing device, guaranteed protection against subtractive attacks is considered extremely difficult. All that we can hope is that the analysis required to remove the watermark is extensive enough that the attacker finds it too costly. For the Branch-Based algorithm the robustness against a subtractive attack is partially based on the number of converted branches which contribute to the fingerprint calculation. Since the algorithm requires the branches to be on a deterministic path the number of usable branches is restricted.

During preliminary development there was question if there would be enough branches on the deterministic path to make the technique a viable option. Through analysis of a variety of different applications, we found a satisfactory number of
<table>
<thead>
<tr>
<th>Program</th>
<th>Branches</th>
<th>Original (S₀)</th>
<th>Watermarked (S₁)</th>
<th>Increase (S₁/S₀)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>39</td>
<td>102,400</td>
<td>106,060</td>
<td>3.57%</td>
</tr>
<tr>
<td>vpr</td>
<td>532</td>
<td>208,896</td>
<td>219,324</td>
<td>4.99%</td>
</tr>
<tr>
<td>gcc</td>
<td>2280</td>
<td>1,646,502</td>
<td>1,682,120</td>
<td>2.16%</td>
</tr>
<tr>
<td>mcf</td>
<td>14</td>
<td>65,536</td>
<td>69,492</td>
<td>6.04%</td>
</tr>
<tr>
<td>crafty</td>
<td>118</td>
<td>303,104</td>
<td>311,508</td>
<td>2.87%</td>
</tr>
<tr>
<td>parser</td>
<td>154</td>
<td>155,648</td>
<td>162,088</td>
<td>4.14%</td>
</tr>
<tr>
<td>eon</td>
<td>1508</td>
<td>425,984</td>
<td>451,152</td>
<td>5.91%</td>
</tr>
<tr>
<td>perlbench</td>
<td>627</td>
<td>737,280</td>
<td>745,928</td>
<td>1.17%</td>
</tr>
<tr>
<td>gap</td>
<td>393</td>
<td>835,659</td>
<td>847,971</td>
<td>1.47%</td>
</tr>
<tr>
<td>vortex</td>
<td>519</td>
<td>606,208</td>
<td>616,440</td>
<td>1.69%</td>
</tr>
<tr>
<td>bzip2</td>
<td>51</td>
<td>86,016</td>
<td>88,960</td>
<td>3.42%</td>
</tr>
<tr>
<td>twolf</td>
<td>341</td>
<td>311,906</td>
<td>318,260</td>
<td>2.14%</td>
</tr>
</tbody>
</table>

Table 9.3: The expected effect of watermarking on program size using a minimal perfect hash function in the native code version of the Branch-Based algorithm.

conditional and unconditional branch instructions. Table 9.4 shows the total number of branches and the number of usable branches in the SPECint-2000 benchmark applications. (Due to the limitations of our disassembly tools we were unable to analyze all of the functions in the applications, thus the overall number of branch instructions could be higher.) By additionally using conditional branches we are able to significantly increase the number of usable branches. This makes the algorithm a viable option.

To successfully perform a subtractive attack the attacker has to perform an extensive manual analysis. First all call instructions which are converted branch instructions have to be isolated. Then the correct target instructions must be identified. Finally, the call instructions have to be replaced with the correct branch instruction type and correct displacement. The completion of such an attack is inhibited by two factors. First static analysis will be virtually useless. Through the use of the fingerprint branch function some of the proper control flow is no longer obvious. For example Figure 9.11 demonstrates how the control flow is interpreted before and after this transformation is applied. Second through the use of integrity checks various forms of dynamic analysis can be prevented.
Figure 9.11: Control flow interpretation before and after the watermark transformation.
Table 9.4: Total number of branches versus the number of usable branches in the SPECint-2000 benchmark suite applications for those functions which we were able to analyze.

In a distortive attack a series of semantics-preserving transformations are applied to the program in an attempt to render the watermark useless. It is the goal of the attacker to distort the software in such a way that the watermark becomes unrecoverable but the program’s functionality and performance remain intact. The original branch function obfuscation is able to provide minimal tamper detection. Since a table is used to store displacements, any transformation applied to a function which alters the displacement between a branch and its target causes the branch function to return an incorrect instruction. Through the use of the integrity checks we were able to extend the tamper detection capabilities throughout the entire program.

To verify our hypothesis that the Branch-Based watermarking algorithm would be resistant to distortive attacks we subjected the benchmark applications to five different obfuscations:

1. Conversion of unconditional jumps to conditional jumps through the use of opaque predicates.

2. Conversion of unconditional jumps to calls to a branch function [64].
3. Conversion of function calls to calls to a branch function [64].

4. Basic block reordering.

5. Merging of two functions into one function whose control flow is regulated through opaque predicates.

In each case the resulting application was non-functional because the integrity checks detected the modification.

For an additive attack to be successful the program has to continue to function properly after the embedding of the second watermark. To simulate an additive attack we double watermarked the benchmark applications using the Branch-Based watermarking algorithm. In each case the result was an improperly functioning application. We believe that a similar result would be obtained if any of the currently known watermarking algorithms were used as the second watermark, however this hypothesis is untested.

The most crucial attack on a fingerprinted application is the collusive attack. With previous watermarking algorithms prevention of a collusive attack is addressed through the use of code obfuscation. The general idea is to apply different sets of obfuscations to the fingerprinted programs. This will make the programs differ everywhere. This is a viable option to thwart a collusive attack, however, it may not always be feasible due to the size and/or performance overhead incurred through obfuscation.

The Branch-Based watermarking scheme is resistant to the collusive attack without the use of obfuscation. The only difference between two fingerprinted programs is the order of the values in the table. Thus, an attacker would have to examine the data section in order to even notice a difference.

The algorithm is still susceptible to dynamic collusive attacks but some of those attacks can be warded off through the use of integrity checks which can recognize the use of a debugger and cause the program to fail. In a dynamic attack the only difference the adversary is going to notice is the value of the key generated at each stage which will ultimately yield a different table slot. In order for an adversary
to launch a successful collusive attack extensive manual analysis in the form of a subtractive attack will be required to remove the fingerprint.

9.3.7 Improving Resilience

Two additional features can be incorporated in the Branch-Based watermarking algorithm to increase the strength: integrity check branch functions and additional indirection. Each of these increases the amount of analysis required to remove the authorship and fingerprint marks.

Integrity Check Branch Functions

Integrity check branch functions (ICBF) are based on the same principle as the FBF. The ICBFs are called by replaced branch instructions not used in the fingerprint generation, i.e. branches not on a deterministic path or branches in a function which is not part of the secret input. The important feature of the ICBFs is that each performs a different type of integrity check. This makes it possible to establish a check and guard system similar to that proposed by Chang and Atallah [26]. For instance the ICBFs could be used to verify that the FBF or other integrity checks have not been altered or removed.

The integrity check value, $v_i$, and the branch instruction offset are used as inputs to generate a key for displacement look up. The displacements for the ICBFs are also stored in the same table used by the FBF. The secure one-way function used to generate the key in the ICBF could be the same as that used by the FBF. If so, the authorship mark would appear in multiple locations throughout the program. If instead, different secure one-way functions are used for the ICBFs, additional authorship marks could be embedded in the program, further strengthening the proof of ownership.

From our analysis we believe that if the strength enhancing features are incorporated into the algorithm the removal of the code which generates the fingerprint will be prohibitively difficult. If the attacker is able to identify which sections of code
are generating the fingerprint then he will have to manually analyze the program to identify all of the call instructions which are converted branch instructions. He will then have to identify the correct target instruction and replace the call with the correct branch and displacement. If the adversary only converts those branches responsible for the fingerprint generation and does not also convert the other branches then the program will fail to execute properly. This is because the integrity check branch functions are designed as a check and guard system. One of their duties is to verify that the fingerprint generating branch function has not been altered or removed. Thus, removal of the fingerprint branch function also requires removal of the integrity check branch functions. While this is not entirely impossible, the manual analysis required to accomplish such a task is extensive.

One aspect of the algorithm which makes the analysis more difficult is that through the use of the integrity check branch functions even more of the obvious proper control flow is removed. Continuing with the example from Figure 9.11, Figure 9.12 demonstrates how the control flow is further altered after the integrity check branch function transformation is applied.

**Additional Indirection**

The second strength enhancing feature is to increase the level of indirection. Additional levels of indirection increase confusion and require more extensive analysis for an attacker. Further indirection can be incorporated in the **Branch-Based** watermarking algorithm by rerouting all calls to the integrity and fingerprint branch functions through a single super branch function which transfers execution to the proper branch function. Figure 9.13 illustrate how an additional single level of indirection can be incorporated.

**9.4 The Branch-Based Algorithm – Java Bytecode Implementation**

Due to restrictions placed on the Java language, the implementation of the **Branch-Based** algorithm for Java requires a different approach. The most limiting aspect is
\begin{verbatim}
push ebp
mov ebp, esp
push ecx
mov dword ptr [ebp-01h], 0h
call branch_function
mov eax, [ebp-01h]
add eax, 01h
mov [ebp-00h], eax
cmp dword ptr [ebp-01h], 05h
call branch_function
mov ecx, [ebp-01h]
push ecx
push "this is i"
call branch_function
add esp, 00h
call branch_function
mov esp, ebp
pop ebp
ret
\end{verbatim}

(c) Transformed function.  (d) Transformed CFG.

Figure 9.12: Control flow interpretation before and after the transformation.

the difficulty in modifying the program counter register which is analogous to the return address modification in native code. This makes it impossible to implement the branch function as it was described for the native code implementation. However, we have devised a technique for watermarking Java applications which maintains the essence of the idea through the use of the Java interface and explicitly thrown exceptions.

The **Branch-Based** watermarking algorithm for Java is implemented within the **SANDMARK** framework. The watermark is embedded by modifying the bytecode instructions and then rewriting the application to a new jar file. Analogous to the native code implementation, the watermark embedding and extraction process is performed in several steps: annotation, tracing, embedding, and extraction.
9.4.1 Annotation

In our implementation of the Branch-Based algorithm for Java the annotation phase is automated and performed at the bytecode level. During annotation at least one call to the method mark() is inserted in each non-abstract method in the application including any constructors. In Java, when a constructor references a constructor from a super class, the method call super() must be the first instruction in the method. To accommodate this language requirement, the call instruction is inserted prior to the completion of method execution. Thus, we identify each return instruction in the method and insert the call to mark immediately before the return.

9.4.2 Tracing

The goal of the tracing phase is to identify the set of methods which are executed as the application is run with the secret input sequence. To facilitate the gathering of this information SandMark makes use of Java’s JDI (Java Debugging Interface) framework. During tracing the application is run as a subprocess under debugging. This makes it possible for SandMark to set breakpoints and examine variables. Each time the mark() method is encountered during the tracing run three pieces of information are recorded: the method’s name, signature, and encapsulating class name. This information makes up the method ID. At the end of the tracing run
we have gathered a list of method IDs for each method executed during the trace. The trace will also indicate if a method has been executed more than once through repeated entries of a particular method ID.

9.4.3 Embedding

Once a trace of the application has been recorded the authorship mark and fingerprint generating code can be embedded. To carry out this phase of the watermarking process four inputs are required: the execution trace, the authorship mark, the fingerprint seed, and the program. Using these inputs the embedding process is broken into seven steps:

1. The fingerprint branch function is constructed and added to the application.

2. An interface and \( n \) classes which implement the interface are constructed and added to the application.

3. From each method identified in the trace, a set of branch instructions are identified.

4. The selected branch instructions are replaced with an instruction which invokes the fingerprint branch function.

5. For each method identified in the trace, the exception table is modified.

6. An array is constructed and inserted in the application.

7. The fingerprint mark for the application is calculated.

As a result of this process a new jar file is produced which when executed with the special input sequence will generate the fingerprint. An additional output of this process is the fingerprint mark assigned to the particular application instance.
Fingerprint Branch Function Construction

Due to the difficulty in modifying the program counter register the Java fingerprint branch function (JFBF) uses a different mechanism to transfer execution control to the branch target. In the JFBF the fingerprint generation is linked to program execution by using the key to force a specific type of exception to be thrown. In the Java implementation an interface A is added to the application which defines a method branch. Additionally, n classes A1, A2, ..., An are added which each implement the interface A. The sole purpose of the method branch is to explicitly throw an exception. Within each of the n classes A1, A2, ..., An the method branch throws a different exception. Instead of a displacement table, an array is constructed which stores objects of type A. The key is used to access the array. Using the obtained object the method branch is invoked which causes a specific exception, the branch-target exception, to be thrown. The exception is propagated up to the method which invoked the JFBF. When this occurs the invoking method will find the exception in its exception table and transfer control to the instruction specified. This instruction is the target of the original branch instruction. Figure 9.14 illustrates the change in execution flow after watermarking.
The first step in constructing the JFBF is to select an integrity check procedure. Due to the security features of the Java language it is difficult to construct stealthy, complex integrity checks. However, through the use of reflection we were able to construct a simple integrity check which will detect many semantics-preserving transformations. When the integrity check is triggered a method from the application is selected using the current key value. Using this method we obtain the declaring class. First we concatenate the names of the declaring class, the method, and a field from the class (if one exists). The hash code of this string gives us the value $x$. Next we construct a string from the bytes in the class. Again we compute the hash code of this string to produce the value $y$. The integrity check value is $v_i = x + y$.

The second task of the JFBF is to calculate the next key based on the current key $k_i$ and the integrity check value $v_i$. This aspect of the JFBF is constructed exactly as described for the native code implementation. $k_{i+1}$ is generated using the secure hash function SHA-1 with an input constructed from $k_i$, $v_i$, and the authorship mark:

$$k_{i+1} = SHA1((k_i \oplus AM) || v_i)$$

In order to force an attacker to use dynamic analysis, the mechanism used to transfer execution to the target draws on Java's use of dynamic method binding. During the embedding process an array of type $A$ is constructed, where $A$ is an interface added to the application. In the array we store objects which are a subclass of $A$. Each of the $n$ subclasses $A_1$, $A_2$, $\ldots$, $A_n$ implement the interface $A$. Each subclass of $A$ implements a method $\text{branch}$ which throws a different exception. The array is organized such that the object whose $\text{branch}$ method throws the branch-target exception associated with a particular key is stored in the slot which is the hash of the key.

$$A_j = T[h(k_{i+1})]$$

Using the obtained object, $A a = T[h(k_{i+1})]$, the method $\text{branch}$ is invoked. This causes the branch-target exception to be thrown and execution to transfer to the target instruction. Because the object $a$ is being referenced as type $A$, which is an
interface, the method lookup for \texttt{branch} will be dynamic and occur at runtime.

In summary, the Java version of the FBF is constructed such that it performs the following tasks:

- A simple integrity check producing the value \(v_i\).
- Generation of the next key using the secure hash function, \(k_i, v_i\), and \(AM\).
- Object look-up via \(A a = T[h(k_{i+1})]\) where \(T\) is an array added to the application and \(h\) is a hash function.
- Invocation of the method \texttt{branch} using \(a, a.\texttt{branch}()\)

Once the JFBF is constructed it, as well as supporting functionality such as the integrity check code, are added to the program.

\textbf{Interface and Implementing Class Construction}

Leveraging Java’s dynamic method binding is a key aspect of the algorithm. To do so we add a single interface \texttt{A} and \(n\) classes \texttt{A1, A2, \ldots, An} which each implement \texttt{A}. In the current implementation the interface \texttt{A} is very simple. It specifies a single public method \texttt{branch}:

\begin{verbatim}
public interface A {
    public void branch() throws Exception;
}
\end{verbatim}

The implementing classes \texttt{A1, A2, \ldots, An} are also very simple. They implement the interface \texttt{A} without adding any additional functionality. The number of implementing classes added is user specified with the default being five and a current maximum of 17. The sole purpose of the method \texttt{branch} is to explicitly throw an exception and each implementation of the method must throw a different exception.

To construct a class \texttt{Ai}, an exception is selected from the pre-composed set of 17 exception types. Figure 9.15 lists the currently used exception types. The set of exception types is pre-composed so as to guarantee that one exception is not a
Figure 9.15: Current list of exception used in the Java implementation of the Branch-Based algorithm.

subtype of another exception in the set. This is an important detail in maintaining proper program behavior. The following class illustrates a typical construction:

```java
class A1 implements A
{
    public void branch() throws java.io.IOException
    {
        throw new java.io.IOException;
    }
}
```

**Branch Instruction Selection**

During the tracing phase the application was executed using the secret input sequence to obtain a trace of the application. The trace identified a set of methods \( M \) through which execution passes. In each method \( m \in M \) the set of branch instructions \( B_m = b_1, \ldots, b_k \) to be converted is selected. As with the native code implementation special care must be taken in selecting which branches are converted. Again we use the dominator set to identify the branches on the deterministic path through the method. This will eliminate the possibility of irregular key evolution.
Selection of the branch instructions in the Java implementation is further constrained by restrictions placed on the exception table entries. In the native code version we were able to replace jmp, jcc, and call instructions. With the Java version we are only able to replace goto and conditional branches. We are unable to replace invoke instructions because the target listed in the exception table must be an instruction within the method.

**Branch Instruction Replacement**

Once we have identified the set of branch instructions $B_m$ for a method $m$ which are to be converted, each $b_i \in B_m$ is replaced with an instruction which will invoke the JFBF. For each branch instruction replaced three mappings are constructed. The first mapping $\psi$, maps the branch instruction to the target instruction ($b_i \rightarrow t_i$). The second mapping $\phi$, maps the branch target to the execution type which will be used in transferring execution control to the target instruction ($t_i \rightarrow e_j$). The third mapping $\theta$, maps the current key $k_i$ to the class whose branch method throws the selected exception type ($k_i \rightarrow A_j$). Just like in the native code implementation, the key must be calculated exactly as it would during application execution. Thus, the branch instructions are replaced in execution order by using the dominator set for the exit block.

The evolution for each method begins with the initial key $k_0$. The value of $k_0$ is derived from the seed. For each method the value of $k_0$ must be slightly different so that each branch-target exception is associated with a unique key. Since the simple integrity check uses the current key value to select a method in the application, we are unable to force a unique key evolution for each method without starting with a unique key. To derive a unique key $k_0$ for method $m_i$ we have to add an additional parameter $p_i$ to $m_i$. This parameter is a unique value which we add to the seed. Thus $k_0^{m_i} = seed + p_i$.

As we replace the branches in execution order the key is evolved following the first two steps of the JFBF:
• We calculate the value $v_i$ by performing the simple checksum integrity check.

• We generate the next key $k_{i+1}$ using the secure hash function SHA-1, $k_i$, $v_i$, and $AM \ (k_{i+1} = SHA1([k_i \oplus AM] || v_i))$.

For each $m \in M$ three mappings are maintained which are used in future steps of the embedding. First we maintain a mapping between the branches and the targets.

$$\psi_m = \{b_1^m \rightarrow t_1^m, \ldots, b_j^m \rightarrow t_j^m\}$$

Second we maintain a mapping between the targets and the selected exception types.

$$\phi_m = \{t_1^m, \ldots, t_j^m\} \rightarrow \{e_1, \ldots, e_n\}$$

Finally, we maintain a mapping between the calculated keys and $n$ classes which implement the interface $A$.

$$\theta_m = \{k_1^m, \ldots, k_j^m\} \rightarrow \{A_1, \ldots, A_n\}$$

For these mappings to be valid the following must hold: if $t_i^m \rightarrow e_l$ and $k_i^m \rightarrow A_q$ where $A_q$'s branch method throws the exception $e_q$ then $e_l = e_q$.

**Exception Table Modification**

In Java every method that catches exceptions has an exception table. The exception table has one entry for each exception caught by a try block. Each entry consists of four pieces of information: the start and end offsets for the instructions which could throw the exception, the start offset where the exception is caught, and the type of exception which is being caught. For example, consider the simple method in Figure 9.16 which has one try and two catch blocks. When we examine the corresponding bytecode we see that the exception table has two entries. The first corresponds to the FileNotFoundException with start offset 0, end offset 23, and the target at bytecode offset 26. The second corresponds to the IOException. Since
both exceptions correspond to the same try block, the start is at offset 0 and the end at offset 23, but the target is at offset 55.

During the branch replacement step we constructed three mappings for each $m \in M$. To facilitate the modification of the exception tables we use two of the mappings:

$$
\psi_m = \{ b_1^m \rightarrow t_1^m, \ldots, b_j^m \rightarrow t_j^m \}
$$

and

$$
\phi_m = \{ t_1^m, \ldots, t_j^m \} \rightarrow \{ e_1, \ldots, e_n \}
$$

Using these two mappings we have all the necessary information to construct new exception table entries. For the start and end offset we use the offset of $b_i$. For the target offset we use the offset of $t_i$. Finally for the exception type we use the exception corresponding to $t_i$. Figure 9.17 illustrates the modifications that would occur if we watermarked the `readLine` method from Figure 9.16. We replaced the `goto` instruction at offset 23 with an `invokevirtual` instruction. Then we added an entry to the exception table corresponding the `CharConversionException` with start and end offset at bytecode offset 23 and a target offset at 67 (the target of the `goto`).

One key aspect of the Java Branch-Based watermarking algorithm is that for each converted branch, $n$ entries must be added to the exception table. One of the entries is the correct target and $n - 1$ are decoys. If the decoy exception entries are omitted, the branch-target pairs become obvious. During the Java verification process exception edges are considered a possible path when checking for consistent stack height, that local variables have been initialized, etc. Thus the targets of the decoy exceptions must be chosen such that the bytecode will still pass the Java verifier.

**Array Construction**

Fingerprint generation and proper program execution are linked by using the current key to force a specific type of exception to be thrown. To facilitate this behavior
Figure 9.16: Illustration of the exception table entries associated with a method which catches exceptions.
```java
Method void readLine(java.lang.String)
    0 new #2 <Class java.io.FileReader>
    3 dup
    4 aload_0
    5 invokespecial #3 <Method java.io.FileReader(java.lang.String)>
    8 astore_1
    9 new #4 <Class java.io.BufferedReader>
   12 dup
   13 aload_1
   14 invokespecial #5 <Method java.io.BufferedReader(java.io.Reader)>
   17 astore_2
   18 aload_2
   19 invokevirtual #6 <Method java.lang.String readLine()> 
   22 astore_3
   23 invokevirtual #15 <Method void fbf()> 
   26 astore_1
   27 getstatic #8 <Field java.io.PrintStream out>
   30 new #9 <Class java.lang.StringBuffer>
   33 dup
   34 invokespecial #10 <Method java.lang.StringBuffer()> 
   37 aload_0
   38 invokevirtual #11 <Method java.lang.StringBuffer append(java.lang.String)> 
   41 ldc #12 <String "Not Found">
   43 invokevirtual #11 <Method java.lang.StringBuffer append(java.lang.String)> 
   46 invokevirtual #13 <Method java.lang.String toString()> 
   49 invokevirtual #14 <Method void println(java.lang.String)> 
   52 goto 67
   55 astore_2
   56 getstatic #8 <Field java.io.PrintStream out>
   59 ldc #16 <String "IO Exception">
   61 invokevirtual #14 <Method void println(java.lang.String)> 
   64 goto 67
   67 return
```

```
Exception table:
   from to  target  type
      0  23     26  <Class java.io.FileNotFoundException>
      0  23     55  <Class java.io.IOException>
      23  23     67  <Class java.io.CharConversionException>
```

Figure 9.17: Illustration of the modification that occur when the `readLine` method is watermarked.
an array of objects is added to the application. In the array we store objects which are a subclass of the interface A, thus a combination of objects A1, A2, ..., An. In this step of the embedding process each of the key-class mappings $\theta_m$ are used to construct a single array $T$. As described in the native code version, the first step in laying out the array is to select a perfect hash function such that each key maps to a unique slot in the array. Again it is best to use a minimal perfect hash function so that the array size is minimized.

$$h = \{k_1, k_2, \ldots, k_m\} \rightarrow \{1, 2, \ldots, n\}, m \leq n$$

The objects are stored in the array such that $T[h(k_i)] = A_j$.

**Fingerprint Mark Calculation**

Similar to the native code version, each method $m \in M$ obtained by executing the application with the secret input sequence, will produce a final method key $k_m$. Each of these keys is combined in a commutative way to produce the fingerprint mark for the application. The variation in the fingerprint mark is obtained through the seed, which is unique for each copy of the application. The generation of the final fingerprint mark is illustrated in Figure 9.9.

**9.4.4 Extraction**

As with embedding, the first step in extracting the fingerprint mark is to execute the application using the secret input sequence. This will cause the fingerprint mark to be built in the fingerprint storage. To extract the fingerprint we access the location where the fingerprint is stored. To extract the authorship mark we simple use static analysis to identify the JFBF. From here we are able to isolate the one-way function to extract the authorship mark.
9.4.5 Evaluation

In this section we provide a complete evaluation of the Java version of the Branch-Based algorithm using the SANDMARK Tool and the evaluation framework described in Chapter 5. Based on this evaluation framework we will be able to make comparisons between the strength of the Branch-Based algorithm and the algorithms presented in Chapters 7 and 8.

Credibility

Like the native code implementation, the Java version uses a blind recognizer to extract the authorship and fingerprint marks. Because we are guaranteed to recognize the marks in a watermarked but otherwise unaltered program, the algorithm provides a high level of credibility.

Data-Rate

The data-rate of the Java version of the Branch-Based algorithm is similar to the native code implementation. The authorship mark is embedded by adding code to the application, thus any size authorship mark can be embedded in any application. The size of the fingerprint mark is influenced by the choice of secure one-way function and the number of methods in the trace. Since SHA-1 is used for this implementation, the fingerprint mark is guaranteed to be at least 160-bits. As the size of the trace increases the size of the fingerprint mark could also increase.

Stealth

The Branch-Based algorithm was not designed to try and hide the marks from an attacker. Instead we incorporated features to increase the robustness of the algorithm. With that in mind we can still evaluate the algorithm with respect to stealth. Overall, the algorithm demonstrates a relatively low level of stealth. The major factor which compromises the stealth is the number of instructions throughout the watermarked application which invoke the same method.
Overhead

The majority of the size impact incurred by the Java version of the Branch-Based algorithm is a result of adding the interface, the \( n \) classes, the array, and the new execution table entries. Because of the way the algorithm is implemented, the size of the watermarked application is not directly linked to the size of the authorship mark or the fingerprint mark.

We watermarked the CaffeineMark and Java Grande applications using 32-, 64-, 128-, 256-, and 512-bit watermarks using various seed values. In each case the increase in program size was the same. What did cause a variation in the size of a watermarked program was the number of classes added to the application. For each additional class the watermark program increased by approximately 400 bytes. When we set the number of classes to five we saw a 58% increase in program size for CaffeineMark and a 3% increase for Java Grande.

The impact on performance varies not with the size of the watermark, but with the number of branch instructions that are replaced. In the case of CaffeineMark and Java Grande we found that only a few instructions could actually be replaced. Because of this no observable slowdown was recorded for either application. We anticipate that as the number of replaced branches increase a rather noticeable slowdown will be observed.

Robustness

Like the native code version, the robustness against subtractive attacks is partially based on the number of converted branches which contribute to the fingerprint calculation. Not only is the number of usable branches restricted by the deterministic path requirement, but also by the type of instructions which can be used. Due to the requirement that the target listed in the exception table must be an instruction in the method, we are unable to replace invoke instructions. These restrictions limit us to goto and conditional branch instructions. Table 9.5 shows that the number of usable branches in our test applications is quite low. It is important to keep in
mind that even though a branch has been identified as usable it may not reside in a function identified by the trace and thus will not be used in the watermarking process.

Robustness against the subtractive attack is also based on the type and strength of the integrity check. Due to the security features of the Java language we have yet to construct an integrity check which will provide the same level of protection against various forms of dynamic analysis that we achieved in the native code version. Overall, the Java version demonstrates a much lower level of resistance to subtractive attacks than the native code implementation.

While the overall strength of the Java version of the Branch-Based algorithm is not as high as the one aimed at native code, we are still able to protect against the distortive attack. However, the protection is only through the integrity checks. The exception table will not provide the same protection as the displacement calculation because a good obfuscation will fix the exception table so as to not risk altering the program behavior. To demonstrate that the Branch-Based algorithm is resistant to distortive attacks we applied the 35 SANDMARK obfuscations to watermarked versions of CaffeineMark and Java Grande. In each case the resulting application was non-functional because the integrity checks detected the modification.

For the same reason that the Branch-Based algorithm is resistant to distortive attacks, it is resistant to additive attacks. Any change in the static code will be

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>Usable</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>172</td>
<td>1</td>
</tr>
<tr>
<td>fft</td>
<td>72</td>
<td>3</td>
</tr>
<tr>
<td>illness</td>
<td>457</td>
<td>46</td>
</tr>
<tr>
<td>lu</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>machineSim</td>
<td>703</td>
<td>47</td>
</tr>
<tr>
<td>matrix</td>
<td>113</td>
<td>2</td>
</tr>
<tr>
<td>probe</td>
<td>98</td>
<td>3</td>
</tr>
<tr>
<td>Java Grande</td>
<td>1761</td>
<td>86</td>
</tr>
<tr>
<td>CaffeineMark</td>
<td>325</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 9.5: Total number of branches versus the number of usable branches in the set of Java test applications.
detected by the integrity checks at some point during the execution. To confirm this level of resistance we applied a second watermark using the ten watermark algorithms from Chapter 7 which have been implemented in the SANDMARK framework. In each case the result was an improperly functioning application.

A collusive attack should be slightly easier in the Java version. The main reason is that the array of objects is in the code as opposed to a less obvious data section. However, outside of where the array, $T$, is initialized two differently fingerprinted programs will be the same.

9.5 Summary

In this chapter we described a novel approach to software watermarking, Branch-Based watermarking, which incorporates ideas from code obfuscation and tamper detection to increase robustness against determined attempts at discovery and removal. Our technique simultaneously provides proof of ownership and the capability to trace the source of the illegal redistribution. This is an improvement over previous techniques which required the developer to choose between the protection techniques. Additionally, the Branch-Based watermark provides a solution for distributing pre-packaged, fingerprinted software which is uniquely linked to the purchaser.

The Branch-Based watermark prototype demonstrates that the technique can successfully thwart both additive and distortive attacks. The technique is also highly resistant to subtractive and collusive attacks, especially for the native code version. Previous fingerprinting techniques addressed the prevention of collusive attacks through the use of code obfuscation which introduces additional overhead. The only static variation introduced by the Branch-Based watermark is in the table. This makes it more highly resilient to collusive attacks even without the use of obfuscation. Additionally, the overhead associated with the native code implementation of the technique is quite minimal and should be tolerable for most applications.

The Branch-Based software watermarking algorithm makes several improvements
over previously proposed techniques:

1. The technique simultaneously provides proof of authorship and the ability to trace the source of the illegal distribution.

2. It demonstrates a significantly higher level of resilience to attack without significant overhead.

3. It provides a means for distributing pre-packaged, fingerprinted software which is linked to the consumer.
IV

SOFTWARE BIRTHMARKING
CHAPTER 10

OVERVIEW

Software birthmarking is a lesser known technique used in the battle to detect software theft. Software birthmarks differ from software watermarks in two important ways. First, in order to embed a watermark it is often necessary to add code to the application. In the case of a birthmark, additional code is never needed. Instead birthmarking relies on inherent characteristics of the programs to show that one program is a copy of another. Secondly, birthmarking cannot prove authorship or be used to identify the source of an illegal redistribution. Rather, birthmarks can only confirm that one program is a copy of another. A strong birthmarking system will be able to provide such confirmation even when transformations have been applied to the code by an adversary in an attempt to hide the theft.

In this chapter we present a general overview of software birthmarking. We begin by formally defining a software birthmark, a software birthmarking system, and the properties used in strength evaluation. This is followed by a discussion of techniques related to birthmarking, but which were not designed with a malicious adversary in mind. Finally, we discuss the simultaneous use of birthmarks and watermarks.

10.1 Applications of Software Birthmarking

The main application of software birthmarking is to address the illegal distribution of all or some part of a program. To illustrate the problem we are trying to solve consider the following scenarios.

1. Alice creates a program and sells it to Bob. Bob makes copies and re-sells the program under a new name.

2. Alice creates a program and sells it to Bob. Bob applies a series of semantics-preserving code transformations to the program, makes copies, and re-sells the
program under a new name.

3. Alice creates a program and sells it to Bob. Bob removes a module to use in his own similar program to make his program better. He then sells the program at a cheaper price.

In each of these scenarios at least part of Alice’s program is contained in Bob’s. If we can identify a large percentage of Alice’s program in Bob’s then we are able to show that Bob copied Alice’s, even if he made some changes or additions. To do this we use software birthmarking

10.2 Definition

One of the earliest uses of the term birthmark in the context of program identification was by Grover [48]. The term was used to mean characteristics occurring in the program by chance which could be used to aid in program identification. A birthmark is distinguished from a fingerprint in that the characteristics used in the fingerprint are intentionally placed in the code, while a birthmark is based on characteristics which are inherent to the code. To formalize we define a software birthmark.

Definition 12 (Software Birthmark). Let \( p \) be a set of modules and \( \text{extract} \) be a method for extracting a set of characteristics from \( p \). Then the birthmark \( b \) is the set of characteristics obtained through the extraction method: \( \text{extract}(p) \rightarrow b \).

The definition of a birthmarking system is based on a measure of similarity between the birthmarks extracted from two sets of modules. In order to hide the fact that copying has taken place an attacker might apply semantics-preserving transformations to the stolen modules. For example, all of the identifiers in the modules might have been renamed; an optimizing register allocator might have been applied so that the stolen and original modules now have different register assignments; or bogus code could be added that is never actually executed. Ideally, the method used to extract a birthmark is resistant to transformation. However, in the event that a
birthmark has been altered by a semantics-preserving transformation we would like to determine the level of similarity. Thus for each method of extraction, a method for comparing two birthmarks must be specified. This leads to the definition of a software birthmarking system:

**Definition 13** (Software Birthmarking System). Given a set of modules $p$ and a set of modules $q$ a software birthmarking system consists of the following functions:

- $extract(p) \rightarrow b_p$
- $extract(q) \rightarrow b_q$
- $similarity(b_p, b_q) \rightarrow [0, 1]$

Analogous to a software watermarking system, a birthmarking system can be characterized based on the code properties used during extraction. A static birthmark is extracted from the statically available information in the program text such as the types or initial values of the fields. A dynamic birthmark is built from information available at execution time. A dynamic algorithm typically works at the program level whereas a static algorithm can target an entire program or individual modules within the program. The same distinction is true with static and dynamic watermarking algorithms. A dynamic algorithm can provide evidence if an entire program is stolen and a static algorithm can be used at different levels of granularity. Definitions 12 and 13 define a static birthmark and a static birthmarking system. The following definitions make the necessary extensions to define a dynamic birthmark and dynamic birthmarking system.

**Definition 14** (Dynamic Software Birthmark). Let $p$ be a program and $I$ be an input sequence. Let $extract$ be a method of extracting a set of characteristics from a program by executing it with the given input sequence. Then the birthmark $b$ is the set of characteristics obtained through the extraction method: $extract(p, I) \rightarrow b$.

**Definition 15** (Dynamic Software Birthmarking System). Given programs $p$ and $q$ and the input sequence $I$ a dynamic software birthmarking system consists of the following functions:
extract(p, I) \rightarrow b_p

extract(q, I) \rightarrow b_q

similarity(b_p, b_q) \rightarrow [0, 1]

In the event of semantics-preserving transformations which lead to a similarity measure of less than one we would still like to be able to say that q is a derivative of p and thus theft likely occurred. We say that q is a derivative of p if the external behavior is the same and the similarity between the birthmarks is within epsilon of one. To formalize the idea we define a software derivative which is based on the observable behavior property.

**Property 1** (Observable Behavior). Let p be a set of modules, let q be a set of modules, let dom(p) be the set of input sequences accepted by p and let out(p, I) be the output of p on input I. For each input I ∈ dom(p), the following must be preserved

- dom(p) ∈ dom(q)
- out(p, I) = out(q, I)

**Definition 16** (Software Derivative). Given two sets of modules p and q and a value \( \epsilon < 1 \), we say that q is a derivative of p if:

- the observable behavior property is maintained, and
- \( 1 - \text{similarity}(b_p, b_q) < \epsilon \)

It is important to note that two programs can exhibit the same external behavior and not be derivatives, but they must exhibit the same external behavior if they are derivatives. For example, iterative and recursive versions of the same function will exhibit the same external behavior but one is not a derivative of the other. For a birthmarking technique to be useful it must make this distinction. Using this definition we make the assumption that \( 1 - \text{similarity}(b_p, b_q) \leq 0.2 \) implies a derivative,
0.2 < 1 - \text{similarity}(b_p, b_q) \leq 0.4 is inconclusive, and 1 - \text{similarity}(b_p, b_q) > 0.4 indicates not derivative. Additional user studies will be necessary to determine if these are reasonable values.

10.2.1 Evaluating Software Birthmarks

The strength of a birthmarking technique lies in its ability to distinguish between similar yet independently written programs and its resistance to transformation. Or, more formally:

**Property 2** (Credibility). Let $p$ and $q$ be independently written programs such that. Then we say \text{extract} is a credible measure if $1 - \text{similarity}(b_p, b_q) > \epsilon$.

**Property 3** (Resistance to Transformation). Let $p'$ be a program obtained from $p$ by applying a set of semantics-preserving transformations $T$. Then we say \text{extract} is resistant to $T$ if $1 - \text{similarity}(b_p, b_{p'}) < \epsilon$.

Property 2 addresses the possibility of the birthmark falsely indicating that $q$ is a copy of $p$. This could occur with independently implemented programs which perform the same task. Since it is highly unlikely that two independently implemented algorithms will contain all of the same details, the birthmarking system should be designed to extract those details which are likely to differ.

With the proliferation of tools for code optimization and obfuscation [3, 9, 10, 11], it is highly probable that an attacker will apply at least one transformation prior to distributing an illegally copied program. Ideally, we desire that a birthmarking system be able to detect a derivative even if a transformation has been applied to that program. This issue is addressed through Property 3.

10.3 Evaluation Framework

Both Chapters 11 and 12 contain evaluations of birthmarking techniques based on Properties 2 and 3. To evaluate the credibility of each of the birthmarking techniques two experiments are required. The first is used to evaluate the ability of the
<table>
<thead>
<tr>
<th>Application</th>
<th>total size</th>
<th>total classes</th>
<th>total methods</th>
<th>total fields</th>
<th>max method</th>
<th>avg method</th>
<th>max inherit chain</th>
<th>avg inherit chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>6481</td>
<td>4</td>
<td>26</td>
<td>8</td>
<td>1056</td>
<td>137</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fft</td>
<td>3284</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>386</td>
<td>98</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>wc</td>
<td>3891</td>
<td>2</td>
<td>14</td>
<td>6</td>
<td>261</td>
<td>37.7</td>
<td>2</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 10.1: Characteristics of each of the Java applications used in evaluating the credibility and resistance of the birthmarking techniques. Method and total sizes are given in bytes.

technique to distinguish between two sets of modules that are independently implemented yet accomplish the same task. This independence experiment is performed using two problems: calculating the factorial function and generating Fibonacci numbers. For each of these problems there exists an iterative and a recursive solution. The Java implementation of these programs can be found in Appendix D.

The second credibility experiment is used to evaluate the ability of the birthmarking technique to distinguish between two sets of modules which do not accomplish the same task. In this false positive experiment we use the four applications from the independence experiment plus three additional Java applications which vary in size and complexity. Because we wanted to establish an automated evaluation framework we chose applications which run without human interaction. This criteria is necessary since computation of a dynamic birthmark requires program execution. Table 10.1 lists characteristics of each of the applications and Table 10.2 provides brief descriptions. The false positive experiment is performed by computing the birthmark similarity for each of the 19 possible combinations. There are actually 21 possible combinations, but we did not include the pairings used in the independence experiment since the tests in those pairings accomplish the same task.

To evaluate a birthmarking technique's resistance to transformation we used five code obfuscation/optimization tools: Codeshield [3], Jarg [7], SANDMARK, Smoke-screen [10], and Zelix KlassMaster [11] to automatically transform a test program into a semantically equivalent but textually different program. We used the same set of Java applications as in the false positive experiment.
<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decode</td>
<td>Implements an algorithm for decoding encrypted messages using Shamir’s Secret Sharing scheme.</td>
</tr>
<tr>
<td>fft</td>
<td>Fast Fourier Transform of complex double precision data. Written by Bruce R. Millar (<a href="mailto:bruce.millar@nist.gov">bruce.millar@nist.gov</a>)</td>
</tr>
<tr>
<td>wc</td>
<td>The Java version of the <code>wc</code> unix command which prints the number of bytes, words, and lines in files. Test program from the Kaffe regression suite. Written by Transvirtual Technologies, Inc. (<a href="http://www.tranvirtual.com">www.tranvirtual.com</a>)</td>
</tr>
</tbody>
</table>

Table 10.2: Brief description of each of the applications used in evaluating the credibility and resistance of the birthmarking techniques.

For Codeshield, Smokescreen, and Zelix KlassMaster we apply the highest level of obfuscation provided by the tool. All three of these tools include name obfuscation, the elimination of debugging information, and some type of control flow obfuscation. Additionally, Smokescreen and Zelix KlassMaster support dead code elimination while Zelix KlassMaster supports string encryption. For the SANDMARK tool we constructed six sets of transformations using 34 different obfuscations. The transformation sets were constructed such that each includes at least one control flow and data transformation. Additionally, the sets were constructed so that each of the 9 obfuscations would transform the test application in a different way. The SANDMARK obfuscations used to create each of the six transformations are listed in Table 10.3.

### 10.4 Related Work

Software birthmarking is based on identifying similarities between programs. Using similarity measures it is possible to accomplish a variety of other goals such as plagiarism detection, authorship analysis, and code clone identification. In the context of software, plagiarism is defined to be the act of creating a new program from a program through small alterations which do not require a detailed understanding of the original program. A variety of techniques have been developed to detect
<table>
<thead>
<tr>
<th>Test</th>
<th>SANDMARK Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMARK Test 1</td>
<td>Static Method Bodies, Bludgeon Signatures, Inliner, Constant Pool Reorderer, Pulicize Fields, Dublicate Registers</td>
</tr>
<tr>
<td>SANDMARK Test 2</td>
<td>Static Method Bodies, Method Merger, Branch Inverter, Rename Registers, String Encoder</td>
</tr>
<tr>
<td>SANDMARK Test 3</td>
<td>Class Splitter, Buggy Code, Field Assignment, Reorder Instructions, Promote Primitive Types</td>
</tr>
<tr>
<td>SANDMARK Test 4</td>
<td>Random Deadcode, Dynamic Inliner, Boolean Splitter, Irreducibility, Variable reassigner, Opaque Branch Insertion</td>
</tr>
<tr>
<td>SANDMARK Test 5</td>
<td>Interleave methods, Insert Opaque Predicates, Array Splitter, Block Marker, Merge Local Integers, Split Classes</td>
</tr>
<tr>
<td>SANDMARK Test 6</td>
<td>False Refactor, Overload Names, Reorder Parameters, Objectify, Parameter Aliases, Promote Primitive Registers, Array Folder</td>
</tr>
</tbody>
</table>

Table 10.3: The six sets of transformations constructed using SANDMARK obfuscations. These transformation will be used in evaluating the resistance of a birthmarking technique.

plagiarism in documents and programs [23, 53, 65, 84, 101]. One such example is Moss [13], an automated tool which has successfully been used to detect similarities between programs at the source code level. The technique used to identify similarities is called winnowing [88]. The idea is to split the file into sequences of length k called k-grams. A hash of each k-gram is then computed and a subset of hashes is selected as the document fingerprint. This technique has proven to be quite successful at detecting plagiarism within student programs. Additional similarity measures include attribute counting [81], structure metrics [13, 84, 101], and Kolmogorov complexity [27].

There are limitations to using plagiarism detection tools to combat software theft. First, the tools compute similarity at the source code level whereas software products are usually distributed without source code. In addition, these systems do
not consider semantics-preserving transformations and the effects of decompilation on the formatting of the source code. For example, it was shown by Collberg, et al. [33] that given the source code of a Java application, simply compiling then decompiling will cause Moss to indicate that the original and the decompiled source code are not likely to be copies.

*Code Clone Detection* is another technique which is used to identify similarities in code. A clone is a program fragment that is identical to another fragment but which may contain minor changes. Such code segments are often introduced through code reuse. It is desirable to identify and remove the duplicated code to improve software maintenance costs. Code clones are generally used to identify similar sections of code within a program. However, the idea could easily be applied to identify clones in different programs. A variety of techniques have been suggested for the identification of clones such as matching abstract syntax trees [22] and comparing sections of code while taking into consideration transformations such as variable and function name changes [20]. Again, the drawback to such techniques is that they are applied at the source code level.

Baker and Manber [21] adapt three tools, previously designed to identify similarity in source code and text, to identify similarities in Java bytecode. *siff* is applied to a large collection of files to identify pairs which contain a large number of common blocks. *dup* can be applied to sets of files to identify similar segments despite renaming transformations. The third tool is the UNIX tool *diff*. Of the three tools, *siff* and *diff* cannot be directly applied to the disassembled bytecode but instead requires the bytecode to be processed into a “normal form.” Additionally, the authors acknowledge that their techniques will not withstand obfuscation.

The application of authorship analysis to literature has been studied extensively. One particular use is the analysis of Shakespearean literature. The extension of authorship analysis to software stems from the desire to identify the authors of malicious attacks on computer systems. After an attack, evidence in the form of source code, object files, and executable code is often left behind. Techniques suggested include programming style and layout metrics [60] as well as data structure, algo-
rithm, and library call choices [92]. While authorship analysis techniques can be used on executable code, the techniques do not consider code transformations.

10.5 Birthmarks and Watermarks

One limitation of software birthmarks is that they provide weaker evidence than software watermarks. Birthmarking is only able to say that one program is likely to be a copy of another, not who the original author is or who is guilty of piracy. However, birthmarking can be used in instances where watermarking is not feasible such as applications where code size is a concern and the watermark would insert additional code. Birthmarking can also be used in conjunction with watermarking to provide stronger evidence of theft. One such example is the watermarking algorithm proposed by Stern, et al. [93] which provides a probability that a specific watermark is contained in the program. If the watermarking algorithm does not 100% guarantee that the watermark is contained in the program, then a birthmark could be used as additional evidence of theft. There are also instances where watermarks fail, e.g. when an attacker is able to apply an obfuscation which destroys the watermark. In these instances a birthmark may still be able to provide proof of program theft since the birthmark may be more resilient to transformations. In Chapter 12, we will present a birthmarking technique which is able to provide proof of theft when many watermarking systems fail.
CHAPTER 11

PUBLISHED BIRTHMARKING TECHNIQUES

Very few software birthmarking techniques have been developed. In this chapter we will present and evaluate two existing static birthmarking techniques. This chapter will serve as a basis when we evaluate the novel dynamic birthmarking technique presented in Chapter 12.

11.1 TaNaMM Birthmark

Tamada, et al. [94, 95] proposed the first software birthmarking technique specifically designed to address the issue of software theft. Their technique, a static birthmarking technique specific to Java classfiles, is composed of four individual birthmarks: constant values in field variables (CVFV), sequence of method calls (SMC), inheritance structure (IS), and used classes (UC). When used in conjunction we refer to the birthmark as the TaNaMM birthmark after the authors Tamada, Nakamura, Monden, and Matsumoto. To illustrate the four birthmarks, the class in Figure 11.1 will be used as a running example.

Many Java classes contain field variables which are initialized to a specific constant value. Initial values cannot be blindly modified without later taking into consideration the changes. This makes Java fields possible candidates for birthmark construction. The CVFV birthmark extracts type and initial value information about the variables declared in the class. For each variable, the type \( t_i \) is extracted along with the initial value \( a_i \). In the event that a field is not assigned an initial value, \( a_i \) is recorded as null. The overall birthmark is the sequence of (type, value) pairs: \( (t_1, a_1), ..., (t_n, a_n)) \). The CVFV birthmark for the class in Figure 11.1 is:
Figure 11.1: A class from the SandMark tool. The class constructs a panel that accepts an Image to use as the skin.
One of the strengths of the Java language is its extensive standard libraries. Because of this, many user methods will make use of these well-known classes. The SMC birthmark builds on the assumption that programmers will consistently use the functionality of well-known classes by examining the sequence of method calls as they appear in the class. The method calls are not necessarily in execution order. Because it is easy to change the names of methods within the application, only those method calls which are in a set of well-known classes, e.g. J2SDK, are considered in the sequence. The SMC birthmark for the class in Figure 11.1 is:

```java
javax.swing.JPanel.<init>()
java.awt.Color.<init>(int),
java.awt.Toolkit java.awt.Toolkit.getDefaultToolkit(),
Class Object.getClass(),
ClassLoader Class.getClassLoader(),
java.net.URL Class.getClassLoader(String),
java.awt.Image java.awt.Toolkit.getImage(java.net.URL),
java.awt.MediaTracker.<init>(java.awt.Component),
void java.awt.MediaTracker.addImage(java.awt.Image, int),
boolean java.awt.MediaTracker.waitForAll(long),
RuntimeException.<init>(Throwable),
int java.awt.Image.getWidth(java.awt.image.ImageObserver),
int java.awt.Image.getHeight(java.awt.image.ImageObserver),
void javax.swing.JPanel.paintComponent(java.awt.Graphics),
boolean java.awt.Graphics.drawImage(java.awt.Image, int,
    int, java.awt.image.ImageObserver)
```

Java is an object oriented language and thus every class has an inheritance structure which can be traced back to java.lang.Object. IS traverses the inheritance structure of the class back to java.lang.Object, again only considering those classes within the set of well-known classes. If a class not in the well-known set is encountered, null is inserted in the sequence in its place. The IS birthmark for the class in Figure 11.1 is:
As was previously mentioned, when programming in Java, it is common to use well-known library classes to implement new functionality. This can be accomplished through inheritance, implementing interfaces, or using these classes as return, argument, or field types. The UC birthmark examines all classes which are used by a given class in any capacity. All classes in the set of well-known classes are included in the sequence, which is then arranged in alphabetical order. The UC birthmark for the class in Figure 11.1 is:

```
java.awt.Color,
java.awt.Component,
java.awt.Dimension,
java.awt.Graphics,
java.awt.Image,
java.awt.image.ImageObserver,
java.awt.MediaTracker,
java.awt.Toolkit,
java.lang.Class,
java.lang.ClassLoader,
java.lang.Object,
java.lang.RuntimeException,
java.lang.String,
java.lang.Throwable,
java.net.URL,
java.swing.JButton,
java.swing.JPanel,
```

### 11.1.1 Similarity of TaNaMM Birthmarks

Each of the individual TaNaMM birthmarks are expressed as a sequence. Thus for two classfiles, \( p \) and \( q \), the corresponding birthmarks are \( b_p = (p_1, ..., p_n) \) and \( b_q = (q_1, ..., q_n) \). Two classes are declared the same if and only if \( p_i = q_i \) for all \( i \), \( 1 \leq i \leq n \). Because it may be possible to alter the birthmark through semantics-preserving code transformations, Tamada et al. define the following similarity measure:
Definition 17 (TaNaMM Birthmark Similarity). Let $b_p = (p_1, ..., p_n)$ and $b_q = (q_1, ..., q_n)$ be birthmarks with length $n$, extracted from classfiles $p$ and $q$. Let $s$ be the number of pairs of $(p_i, q_i)$'s such that $p_i = q_i$ ($1 \leq i \leq n$). Then, 
$$\text{similarity}(b_p, b_q) = \frac{s}{n}.$$  

The major drawback to this similarity measure is that a code transformation could alter the birthmark by increasing the sequence length. When this occurs, it is unclear how to calculate the similarity. A straight-forward extension of the similarity measure would be to disregard the “extra” values. For example, let $b_p = (p_1, ..., p_m)$ and $b_q = (q_1, ..., q_n)$ where $m < n$. The similarity measure would compute $s$ such that each $q_i$ for $m < i \leq n$ are disregarded. It is obvious that such a measure could yield lower similarity results that do not accurately identify classfile theft. To address this limitation we propose two similarity measures which more reliably identify copies. Depending on the TaNaMM birthmark, similarity can be based on unordered containment or the longest common subsequence.

For the CVFV birthmark the order in which the fields are declared in the class is unimportant. It is trivial for an attacker to rearrange the fields, thus, the birthmark similarity measure should not be restricted to an ordered sequence. Additionally, the measure should account for insertions and deletions since it is also trivial for an attacker to insert new fields. Based on these aspects, we propose the following measure of similarity be used for the CVFV birthmark.

Definition 18 (CVFV Similarity). Let $b_p = (p_1, ..., p_m)$ and $b_q = (q_1, ..., q_n)$ be birthmarks extracted from classfiles $p$ and $q$ where $m \leq n$. Then, 
$$\text{similarity}(b_p, b_q) = \frac{|b_p \cap b_q|}{m}.$$  

The remaining three TaNaMM birthmarks are all defined such that the order of the values is important. Thus, the similarity measure must preserve this property. However, the measure should also consider insertions and deletions which could occur through various code transformations such as inserting bogus code or dead code removal. These requirements can be modeled using the longest common subsequence (LCS) problem. The LCS between two sequences is defined to be the
longest subsequence common to both sequences. Therefore, the similarity measure for SMC, IS, and UC should be defined as follows:

**Definition 19 (SMC, IS, and UC Similarity).** Let \( b_p = (p_1, \ldots, p_m) \) and \( b_q = (q_1, \ldots, q_n) \) be birthmarks extracted from classfiles \( p \) and \( q \) where \( m \leq n \). Then, 

\[
\text{similarity}(b_p, b_q) = \frac{|LCS(b_p, b_q)|}{m}.
\]

The four birthmark techniques were designed to be used in conjunction. So, given classfiles \( p \) and \( q \), we can conclude that \( q \) is a derivative of \( p \) if and only if \( 1 - \text{similarity}_{\text{CVFV}}(b_p, b_q) < \epsilon \), \( 1 - \text{similarity}_{\text{SMC}}(b_p, b_q) < \epsilon \), \( 1 - \text{similarity}_{\text{IS}}(b_p, b_q) < \epsilon \), and \( 1 - \text{similarity}_{\text{UC}}(b_p, b_q) < \epsilon \). Likewise, overall similarity is based on the combination of the individual similarities. Tamada et al. do not define exactly how the overall similarity is computed from the individual similarities. A reasonable method would be to take the average. This is the technique we use in the evaluation which follows.

### 11.1.2 Evaluation

The TaNaMM birthmark works well given two classfiles, but the strength suffers when given two collections of classfiles. This is because three of the four individual birthmarks are order dependent, that is, they rely on the order of the characteristics as they are statically laid out in the classfile. For example, consider the SMC birthmark, i.e. the sequence of method calls in the classfile. The order is easy to manipulate by simply rearranging the methods in the class. Performing such a transformation would cause the birthmark to indicate a lower similarity. Only UC addresses this issue by using an alphabetic ordering of the classes. In addition, after an obfuscation has been applied it is not always clear which original class to compare with which obfuscated class. To handle this problem, every class in the original must be compared with every class in the obfuscated program, which for large programs may not be a feasible option.

To evaluate the TaNaMM birthmark we used the evaluation framework detailed in Chapter 10. The similarity measure differs from that used in previous evaluations
Table 11.1: The TaNaMM birthmark has a higher level of resistance to transformation using Definitions 18 and 19. birthmark’ indicates the Tamada et al. definition of similarity was used.

(e.g. [94, 95, 74, 76]) and is instead based on Definitions 18 and 19, which we will show more accurately measures similarity in the presence of semantics-preserving transformations. It is important to note that the evaluation was conducted at the program level without manual intervention. This is not ideal for computing similarity using the TaNaMM birthmark. However, it is more representative of the environment in which theft will be detected since it may not be clear which classes to compare due to changes in class names or structure.

The TaNaMM birthmark was previously evaluated using the similarity measure defined by Tamada et al. (Definition 17) [74]. Using the new similarity measure defined in Definitions 18 and 19 the TaNaMM birthmark demonstrates a higher level of resistance to transformation. Table 11.1 shows the results previously obtained when the obfuscators Codeshield, Smokescreen, and Zelix KlassMaster were applied to the testcase wc and the results we obtained using the new similarity measure. These results show that for two of the three obfuscations the TaNaMM birthmark moves from reporting an inconclusive result ([0.6 — 0.8]) to indicating a derivative was found ([0.8 — 1]). For the case of Codeshield in which TaNaMM originally reported a derivative was found, the level of similarity greatly increased using the new similarity measure. Because the similarity measures in Definitions 18 and 19 produces a higher level of similarity (ie the values are closer to 1) we believe they provide a more accurate representation of the TaNaMM birthmarking technique than the original similarity measure.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>CVFV</th>
<th>CVFV</th>
<th>SMC</th>
<th>SMC</th>
<th>IS</th>
<th>IS</th>
<th>UC</th>
<th>UC</th>
<th>TaNaMM</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codeshield</td>
<td>.83</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.45</td>
<td>1</td>
<td>.82</td>
<td>.96</td>
<td></td>
</tr>
<tr>
<td>Smokescreen</td>
<td>.83</td>
<td>.83</td>
<td>.16</td>
<td>.66</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.75</td>
<td>.87</td>
<td></td>
</tr>
<tr>
<td>KlassMaster</td>
<td>.67</td>
<td>1</td>
<td>.25</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.73</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Table 11.2: Results from the independence experiment using the TaNaMM birthmark techniques.

**Credibility**  In the independence experiment, the TaNaMM birthmarks were evaluated to determine if they would indicate two independently implemented applications that accomplish the same task were not derivatives. Table 11.2 shows that all four techniques indicated the iterative and recursive versions of factorial and Fibonacci were derivatives. When combined the TaNaMM birthmarking technique also indicates derivative for both factorial and Fibonacci. (The similarity results for all experiments can be found in Appendix E.)

Tamada et al. [94] do state that their birthmarking technique is unable to distinguish between independently written applications which are small. However, their techniques are aimed at class level theft and classes are often designed to be specific and minimal. In Collberg et al. [34] we analyzed 1132 Java applications obtained from the Internet. In this analysis we found that, on average, a class contains 4.4 fields. 21.8\% of the field types are int and 15\% are java.lang.String. The average height of an inheritance graph for an application is 4.5 but the average class is only at a depth of 2.1. Additionally, 47.1\% of classes extend java.lang.Object directly. The average class only contains nine methods and the average number of bytecode instructions per method is only 33. This analysis indicates that, on average, classes are small and rely on a limited inheritance structure. Thus, the TaNaMM birthmark may have difficulty distinguishing between many independently written classes.
Table 11.3: Results from the false positive experiment using the TaNaMM birthmarking techniques.

For the false positive experiment we applied the TaNaMM birthmark to 19 different pairings of the seven Java applications. Of the four individual birthmarks CVFV performed the worst. Using this birthmark only one pairing was accurately marked as not being derivatives. The IS results were only minimally better with one pair flagged as inconclusive and one not derivatives. The UC results were not much better with one inconclusive and two not derivative results. Finally, SMC had the best results with three pairings marked inconclusive and three not derivatives. Unfortunately, when combined the TaNaMM birthmark indicates 16 of the pairings are derivatives, two inconclusive, and only one accurately marked as not derivative. The results from the false positive experiment are shown in Table 11.3. Based on the results from the two credibility experiments we conclude that the TaNaMM birthmarking technique demonstrates a very low credibility.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>IterFact</th>
<th>IterFib</th>
<th>RecurFact</th>
<th>RecurFib</th>
<th>decode</th>
<th>fit</th>
<th>wc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SmokeScreen</td>
<td>⊕</td>
<td>⊕</td>
<td></td>
<td></td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>Codeminder</td>
<td>⊕</td>
<td>⊕</td>
<td></td>
<td></td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>Zeex Klassmaster</td>
<td>⊕</td>
<td>⊕</td>
<td></td>
<td></td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>jarg</td>
<td>⊕</td>
<td>⊕</td>
<td></td>
<td></td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
</tbody>
</table>

⊕ : derivative  
± : inconclusive  
⊖ : not derivative

Table 11.4: Results from the resistance experiment using the TaNaMM birthmarking technique.

Resistance to Transformation We also evaluated the TaNaMM birthmark with respect to resistance to semantics-preserving transformations. This evaluation was conducted by applying the ten obfuscation/optimization tests to the seven Java applications. From Table 11.4 it can be seen that in each case the TaNaMM birthmarking technique accurately flagged the transformed program as a derivative of the original. Thus, the TaNaMM technique demonstrates a high level of resistance to transformation for this set of testcases.

11.2 K-Gram Based Birthmark

The K-Gram birthmark [76] is a static technique based on instruction sequences which we proposed as a language independent alternative to the TaNaMM birthmark. A k-gram is a contiguous substring of length k which can be comprised of letters, words, or in the case of executable programs, opcodes. The k-gram birthmark is based on static analysis of the executable program. For each method in a module we compute the set of unique k-grams by sliding a window of length k over the static instruction sequence as it is laid out in the executable. For example, consider the
method in Figure 11.2. At ② we compute the set of $k$-grams where $k = 2$. Because
the birthmark is constructed of the unique $k$-grams, at ⑤ we remove any duplicates
(in this example <invokevirtual, invokevirtual> occurs twice). This leaves us
with a birthmark of size 9.

The birthmark for the module is the union of the birthmarks of each method
in the module. The order of the $k$-grams within the set is unimportant as is the
frequency of occurrence of each $k$-gram. By using the unique $k$-grams without their
associated frequency the birthmark is less susceptible to semantics-preserving trans-
formations. For example, an obfuscation which duplicates basic blocks will increase
the frequency of the particular $k$-grams in the duplicated block. Additionally, be-
because the birthmark is independent of the order of the methods in the module or the
modules within the program, the technique can be used at the module or program
level. Figure 11.3 illustrates a module level K-Gram birthmark where $k = 3$.

In order for a birthmark to be useful for theft detection it must be based on
a characteristic of the program which will uniquely vary from one program to the
next. Prior to proposing $k$-grams as a basis for a birthmark we investigated whether
a specific set of $k$-grams is unique to a program. In our study we examined the
frequency of $k$-grams, where $1 \leq k \leq 7$, in 222 Java jar-files obtained from the
Internet. The programs ranged in size from 2 to 11,329 methods and 1 to 586 classes.
We are assuming that since these programs were obtained from a variety of sources
that they represent a reasonable random sampling of Java programs. Figures 11.4
and 11.5 show the top 10 most frequently occurring bytecode sequences for each of
the $k$-grams. These figures show that even the top 10 most frequently occurring
$k$-grams have a very low frequency. For $k = 1$ the top sequence has a frequency of
10%, for $k = 2$ the top sequence has decreased to a frequency of 5%, and by $k = 7$
the frequency is down to just over 1%.

Based on this analysis we formed the hypothesis that two independent programs
will have very few $k$-grams in common. To further support our hypothesis we
examined the total number of unique $k$-grams and the total number of $k$-grams in
the 222 programs. We found that as the value of $k$ increases the ratio of unique
Figure 11.2: Java bytecode of a single method and its associated K-Gram birthmark where $k = 2$. 
Figure 11.3: Java bytecode of a classfile which recursively computes the factorial of 15 and its associated K-Gram birthmark where $k = 3$. 
Figure 11.4: Top ten most frequently occurring bytecode sequences of size $1 \leq k \leq 4$.

$k$-grams to total $k$-grams decreases. This also indicates that two program will have few $k$-grams in common for larger value of $k$. Table 11.5 shows the results.

### 11.2.1 Similarity of K-Gram Birthmarks

The K-Gram birthmark is the set of unique opcode sequences of length $k$. Let $b_p = \{p_1, ..., p_m\}$ and $b_q = \{q_1, ..., q_n\}$ be birthmarks of the sets of modules $p$ and $q$ respectively. We say that two sets of modules are the same if and only if $b_p = b_q$, i.e. if $|b_p| = |b_q| = |b_p \cap b_q|$. The proliferation of code obfuscation and optimization tools has made it far more likely that semantics-preserving transformations will be used in an attempt to defeat software theft detection mechanisms. Even in the event of this type of attack we would like to be able to conclude that $q$ is a derivative of $p$.

Broder [24] explores a parallel idea for comparing documents. He defines the two mathematical notions of resemblance and containment in order to quantitatively measure the similarity of two documents. The resemblance of two documents, $p$ and
<table>
<thead>
<tr>
<th>Bytecode Sequence</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>dup sipush iconst_m1 bastore dup</td>
<td>1.35%</td>
</tr>
<tr>
<td>bastore dup sipush iconst_m1 bastore</td>
<td>1.35%</td>
</tr>
<tr>
<td>sipush iconst_m1 bastore dup sipush</td>
<td>1.35%</td>
</tr>
<tr>
<td>iconst_m1 bastore dup sipush iconst_m1</td>
<td>1.33%</td>
</tr>
<tr>
<td>bastore dup sipush bpush bastore</td>
<td>0.80%</td>
</tr>
<tr>
<td>dup sipush bpush bastore dup</td>
<td>0.59%</td>
</tr>
<tr>
<td>sipush bpush bastore dup sipush</td>
<td>0.59%</td>
</tr>
<tr>
<td>bpush bastore dup sipush bpush</td>
<td>0.41%</td>
</tr>
<tr>
<td>bastore dup bpush bpush bastore</td>
<td>0.34%</td>
</tr>
<tr>
<td>lastore dup bpush ldc_w lastore</td>
<td>0.33%</td>
</tr>
</tbody>
</table>

(a) $k = 5$

<table>
<thead>
<tr>
<th>Bytecode Sequence</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>dup sipush iconst_m1 bastore dup sipush</td>
<td>1.18%</td>
</tr>
<tr>
<td>bastore dup sipush iconst_m1 bastore dup</td>
<td>1.18%</td>
</tr>
<tr>
<td>iconst_m1 bastore dup sipush iconst_m1 bastore</td>
<td>1.14%</td>
</tr>
<tr>
<td>sipush iconst_m1 bastore dup sipush iconst_m1</td>
<td>1.14%</td>
</tr>
<tr>
<td>bastore dup sipush bpush bastore dup sipush</td>
<td>0.82%</td>
</tr>
<tr>
<td>dup sipush bpush bastore dup sipush</td>
<td>0.82%</td>
</tr>
<tr>
<td>bpush bastore dup sipush bpush bastore</td>
<td>0.73%</td>
</tr>
<tr>
<td>sipush bpush bastore dup sipush bpush</td>
<td>0.73%</td>
</tr>
<tr>
<td>lastore dup bpush ldc_w lastore dup</td>
<td>0.34%</td>
</tr>
<tr>
<td>lastore dup bpush ldc_w lastore dup</td>
<td>0.34%</td>
</tr>
</tbody>
</table>

(b) $k = 6$

<table>
<thead>
<tr>
<th>Bytecode Sequence</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>bastore dup sipush iconst_m1 bastore dup sipush</td>
<td>1.21%</td>
</tr>
<tr>
<td>iconst_m1 bastore dup sipush iconst_m1 bastore dup</td>
<td>1.17%</td>
</tr>
<tr>
<td>dup sipush iconst_m1 bastore dup sipush iconst_m1</td>
<td>1.17%</td>
</tr>
<tr>
<td>sipush iconst_m1 bastore dup sipush iconst_m1 bastore</td>
<td>1.17%</td>
</tr>
<tr>
<td>bastore dup sipush bpush bastore dup sipush</td>
<td>0.81%</td>
</tr>
<tr>
<td>dup sipush bpush bastore dup sipush</td>
<td>0.74%</td>
</tr>
<tr>
<td>sipush bpush bastore dup sipush bpush bastore</td>
<td>0.74%</td>
</tr>
<tr>
<td>sipush bpush bastore dup sipush bpush</td>
<td>0.74%</td>
</tr>
<tr>
<td>lastore dup bpush ldc_w lastore dup</td>
<td>0.35%</td>
</tr>
<tr>
<td>lastore dup bpush ldc_w lastore dup</td>
<td>0.35%</td>
</tr>
</tbody>
</table>

(c) $k = 7$

Figure 11.5: Top ten most frequently occurring bytecode sequences of size $5 \leq k \leq 7$. 
\begin{table}
\begin{tabular}{|c|c|c|}
\hline
Window Size & Unique k-grams & Total k-grams \\
\hline
1 & 197 & 5,416,799 \\
2 & 8727 & 5,281,483 \\
3 & 86,345 & 5,109,318 \\
4 & 310,659 & 4,914,995 \\
5 & 634,551 & 4,765,130 \\
6 & 937,360 & 4,631,019 \\
7 & 1,170,570 & 4,518,672 \\
\hline
\end{tabular}
\caption{Number of unique \textit{k}-grams and total number of \textit{k}-grams for $1 \leq k \leq 7$.}
\end{table}

\( q \), is defined to be:
\[
 r(p, q) = \frac{|f(p) \cap f(q)|}{|f(p) \cup f(q)|}
\]

while the containment of \( p \) within \( q \) is defined as:
\[
 c(p, q) = \frac{|f(p) \cap f(q)|}{|f(p)|}
\]

To decide which measurement we should choose for defining the similarity of two sets of modules, we reflect back on the scenarios from Chapter 10.

1. Alice creates a program and sells it to Bob. Bob makes copies and re-sells the program under a new name.

2. Alice creates a program and sells it to Bob. Bob applies a series of semantics-preserving code transformations to the program, makes copies, and re-sells the program under a new name.

3. Alice creates a program and sells it to Bob. Bob removes a module to use in his own similar program to make his program better. He then sells the program at a cheaper price.

In each of these scenarios at least part of Alice’s program is contained in Bob’s. If we can identify a large percentage of Alice’s program in Bob’s then we are able to show that Bob copied Alice’s, even if he made some changes or additions. Since
resemblance will also consider the additions made by Bob, it is not the correct measure of similarity for detecting theft. The containment measure $c(p, q)$ is therefore the correct quantity for measuring birthmark similarity.

**Definition 20.** (K-Gram Similarity) Let $b_p = \{p_1, ..., p_m\}$ and $b_q = \{q_1, ..., q_n\}$ be K-Gram birthmarks extracted from the two sets of modules $p$ and $q$. The similarity between $b_p$ and $b_q$ is defined by:

$$\text{similarity}(b_p, b_q) = \frac{|b_p \cap b_q|}{|b_p|}.$$  

The above definition of similarity assumes that $p$ is the original and $q$ is the stolen copy. It is often the case that such an assumption is valid. However, in the event that it is unknown which is the original, but it is still desirable to compute the similarity, such as detecting plagiarism in two student programs, then the similarity would be defined as $\max(\text{similarity}(b_p, b_q), \text{similarity}(b_q, b_p))$.

### 11.2.2 Evaluation

As with the TaNaMM birthmark, we evaluated the K-Gram birthmark using the framework described in Chapter 10. In this evaluation we studied the ability of the K-Gram birthmark to satisfy the properties of credibility and resistance to transformation.

**Credibility** An important feature of a birthmarking technique is that it can distinguish between two sets of modules which are independently implemented, even if they accomplish the same task. To demonstrate that the K-Gram birthmarking technique is able to make this distinction we used the independence experiment. Table 11.6 shows that even smaller values of $k$ produced results indicating that the iterative and recursive versions of factorial and Fibonacci are not derivatives. For $k = 2$, the similarity was close to 0.4 for both sets and the value decreased as $k$ increased.
<table>
<thead>
<tr>
<th>Programs</th>
<th>$k = 2$</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

⊕ : derivative  
± : inconclusive  
⊗ : not derivative

Table 11.6: Results from the independence experiment using the K-Gram birthmarking technique and $2 \leq k \leq 8$.

To evaluate the credibility of the K-Gram birthmark with respect to false positives we examined 19 pairings of the seven Java applications. In this experiment we examined if the K-Gram birthmarking technique would accurately indicate that the pairs of programs were not derivatives when $2 \leq k \leq 8$. We found that as $k$ increased, the level of similarity between the pairs decreased. The results in Table 11.7 show that even for $k = 2$, only one pairing was inaccurately flagged as derivatives and two as inconclusive. At $k = 3$ only one pairing is still flagged as derivatives and no pairing are inconclusive. By $k = 7$ all pairings are considered not derivatives. Based on the results from the two credibility experiments we conclude that the K-Gram birthmarking technique demonstrates a high level of credibility.

**Resistance to Transformation** To test the resistance of the K-Gram birthmarking technique we used the ten obfuscation/optimization tests. We computed the similarity using $k$-grams for $2 \leq k \leq 8$. As was expected, as the value of $k$ increased, the accuracy of the K-Gram birthmarking technique decreased.

The tables in Figure 11.6 show that with the exception of the obfuscations SANDMARK Test 3 and SANDMARK Test 6, the K-Gram birthmark was able to indicate the possibility that theft had occurred for smaller values of $k$. For $k = 2$, the technique accurately flagged all seven testcases as derivatives when the transformations SANDMARK Test 1, SANDMARK Test 2, SANDMARK Test 4, Codeshield,
\begin{tabular}{|l|c|c|c|c|c|c|c|}
\hline
Programs & k = 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
IterFact, IterFib & \textcolor{green}{\oplus} & \textcolor{green}{\oplus} & \textcolor{green}{\pm} & \textcolor{red}{\pm} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
IterFact, RecurFib & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
IterFact, decode & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
IterFact, fft & \textcolor{red}{\pm} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
IterFact, wc & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
RecurFact, IterFib & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
RecurFact, RecurFib & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
RecurFact, decode & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
RecurFact, fft & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
RecurFact, wc & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
IterFib, decode & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
IterFib, fft & \textcolor{red}{\pm} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
IterFib, wc & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
RecurFib, decode & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
RecurFib, fft & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
RecurFib, wc & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
decode, fft & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
decode, wc & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
fft, wc & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} & \textcolor{red}{\circ} \\
\hline
\end{tabular}

\begin{itemize}
\item \textcolor{green}{\oplus}: derivative
\item \textcolor{red}{\pm}: inconclusive
\item \textcolor{red}{\circ}: not derivative
\end{itemize}

Table 11.7: Results from the false positive experiment using the K-Gram birthmarking technique and $2 \leq k \leq 8$.

Zelix Klassmaster, and jarg were used. Additionally, for the transformations SandMark Test 5 and Smokescreen no testcase was marked not derivative. When $k$ is increased to three, the technique was still able to accurately flag all the testcases as derivatives for three of the transformations (SandMark Test 2, Codeshield, and jarg). Additionally, for the transformations SandMark Test 1, SandMark Test 4, SandMark Test 5, Smokescreen, and Zelix Klassmaster only three testcases were marked not derivative. The results for $k = 4$ are relatively similar. Thus, for smaller values of $k$, the K-Gram birthmarking technique demonstrates a moderately high level of resistance to transformation. (The resistance results for all seven applications and all seven values of $k$ ($2 \leq k \leq 8$) can be found in Appendix E.)
\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Obfuscation & IterFact & IterFib & RecurFact & RecurFib & decode & ft & wc \\
\hline
SMARK Test 1 & $\oplus$ & $\oplus$ & $\oplus$ & $\oplus$ & $\oplus$ & $\oplus$ & $\oplus$ \\
SMARK Test 2 & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 3 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 4 & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 5 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 6 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
Smokescreen & $\oplus$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ \\
Codenfield & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
Zelex Klassmaster & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ \\
jarg & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
\hline
\end{tabular}
\end{center}

(a) $k = 2$

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Obfuscation & IterFact & IterFib & RecurFact & RecurFib & decode & ft & wc \\
\hline
SMARK Test 1 & $\oplus$ & $\oplus$ & $\oplus$ & $\oplus$ & $\oplus$ & $\oplus$ & $\oplus$ \\
SMARK Test 2 & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 3 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 4 & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 5 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 6 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
Smokescreen & $\oplus$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ \\
Codenfield & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
Zelex Klassmaster & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ \\
jarg & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
\hline
\end{tabular}
\end{center}

(b) $k = 3$

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
Obfuscation & IterFact & IterFib & RecurFact & RecurFib & decode & ft & wc \\
\hline
SMARK Test 1 & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 2 & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 3 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 4 & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 5 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
SMARK Test 6 & $\ominus$ & $\ominus$ & $\ominus$ & $\ominus$ & $\pm$ & $\pm$ & $\pm$ \\
Smokescreen & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ \\
Codenfield & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
Zelex Klassmaster & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ & $\pm$ \\
jarg & $\odot$ & $\odot$ & $\odot$ & $\odot$ & $\pm$ & $\pm$ & $\pm$ \\
\hline
\end{tabular}
\end{center}

(c) $k = 4$

$\oplus$ : derivative  \\
$\pm$ : inconclusive  \\
$\ominus$ : not derivative

Figure 11.6: Results from the resistance experiment using the K-Gram birthing technique and $k = 2, 3, 4$. 
Credibility/Resistance Trade-off  One important characteristic to note about the K-Gram birthmark is that as $k$ increases there is an increase in credibility and a decrease in resistance to transformation. Thus, it is necessary to identify the value of $k$ which maximizes both the credibility and resistance. Through the data generated for the evaluation we are able to form a hypothesis that $k = 3$ or $k = 4$ is an appropriate value of $k$. At $k = 3$, only one pairing in the false positive experiment was inaccurately flagged as derivative. For $k = 4$, this pairing moved from derivative to inconclusive. For $k = 3$, the K-Gram birthmark is still able to detect theft for the majority of the obfuscations. However, at $k = 4$ the detection becomes less reliable for a few of the obfuscations. To further support our hypothesis we formed 106 pairs from the random sampling of Java jar files used in Section 11.2. At $k = 3$, only 5 pairs still had a similarity $> 0.55$ and at $k = 4$ all pairs had a similarity $\leq 0.55$. Thus, based on the sampling of data, $k = 3$ or $k = 4$ provides an appropriate tradeoff between credibility and resilience for the K-Gram birthmark technique. $k = 3$ demonstrates a higher level of resistance to transformation and appears to be the best choice for this data set. Thus, from now on we will use a K-Gram birthmark of size 3 when we compare it to the other birthmarking techniques.

11.3 K-Gram Birthmark vs. TaNaMM Birthmark

Using the similarity measures proposed in Definitions 18 and 19 we were able to significantly improve the strength of the TaNaMM birthmarking technique with respect to resistance to transformation. In a previous evaluation of the two static birthmarking techniques the $k$-gram birthmark performed well above the TaNaMM [76]. As can be seen in Tables 11.8 the TaNaMM technique more reliably indicates the occurrence of theft when considering resistance to transformation.

Even though the TaNaMM birthmarking technique can more reliably detect theft in the presence of semantics-preserving transformations, the credibility results cast doubt on the usability of the technique. Table 11.9 provides a comparison between the credibility results for the TaNaMM birthmark and the $k$-gram birthmark where
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>IterFact</th>
<th>IterFib</th>
<th>RecurFact</th>
<th>RecurFib</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TaNaMM</td>
<td>K-Gram</td>
<td>TaNaMM</td>
<td>K-Gram</td>
</tr>
<tr>
<td>SANDMark Test 1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>±</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>decode</th>
<th>fit</th>
<th>wc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TaNaMM</td>
<td>K-Gram</td>
<td>TaNaMM</td>
</tr>
<tr>
<td>SANDMark Test 1</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>jarg</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

⊕ : derivative  
± : inconclusive  
☐ : not derivative

Table 11.8: Results from the resistance experiment using the TaNaMM and K-Gram birthmarking techniques where \( k = 3 \).
\[
\begin{array}{|l|c|c|}
\hline
\text{Programs} & \text{TaNaMM} & \text{K-Gram} \\
\hline
\text{IterFact, IterFib} & + & + \\
\text{IterFact, RecurFib} & + & + \\
\text{IterFact, decode} & + & + \\
\text{IterFact, fft} & + & + \\
\text{IterFact, wc} & + & + \\
\text{RecurFact, IterFib} & + & + \\
\text{RecurFact, RecurFib} & + & + \\
\text{RecurFact, decode} & + & + \\
\text{RecurFact, fft} & + & + \\
\text{RecurFact, wc} & + & + \\
\text{IterFib, decode} & + & + \\
\text{IterFib, fft} & + & + \\
\text{IterFib, wc} & + & + \\
\text{RecurFib, decode} & + & + \\
\text{RecurFib, fft} & + & + \\
\text{RecurFib, wc} & + & + \\
\text{decode, fft} & \pm & + \\
\text{decode, wc} & + & + \\
\text{fft, wc} & \pm & + \\
\hline
\end{array}
\]

⊕ : derivative  \\
± : inconclusive  \\
⊖ : not derivative

Table 11.9: Results from the false positive experiment using the TaNaMM and K-Gram birthmarking techniques where \( k = 3 \).

\( k = 3 \). From this table it clear that the \( k \)-gram birthmark is far more reliable at distinguishing between unique applications. For 16 of the pairings, the TaNaMM technique incorrectly flags them as derivatives as opposed to the one inaccurate result for the K-Gram technique.
CHAPTER 12

WHOLE PROGRAM PATH BIRTHMARKING

In the previous chapter we presented two published static software birthmarking techniques. Through an evaluation of these techniques we showed that the TaNaMM technique is highly resistant to transformation but has a very low degree of credibility. On the other hand, the K-Gram technique has a high level of credibility but only an average resistance to transformation. Through our study of software watermarking techniques we found that in most cases, watermarks which were embedded using static aspects of the program were less resistant to semantics-preserving transformations. Based on this research we hypothesized that if we identified the right set of dynamic program characteristics we should be able to develop a dynamic birthmarking technique which displays a high level of credibility and is resistant to semantics-preserving transformations.

In this chapter we present a novel dynamic software birthmarking technique which demonstrates the properties of credibility and resistance to transformation under most circumstances. The Whole Program Path (WPP) birthmarking algorithm is a dynamic technique which builds a birthmark from the execution pattern of the program. To construct such a birthmark we draw on the whole program path idea presented by Larus to represent a program’s dynamic control flow in a compressed form [61]. Because the WPP birthmark is dynamic it can only be used to detect application-level theft. However, the technique does not rely on features specific to any language and thus like the K-Gram technique is language independent.

12.1 Whole Program Paths

A WPP birthmark for a program \( p \) is constructed by creating a specific representation of the execution trace of \( p \) when it is executed with the input sequence \( I \).
Unfortunately for the average program, the execution trace can quickly become prohibitively large. To address this issue the execution trace is represented in the whole program path compressed form proposed by Larus. The basic idea is to use the SEQUITUR algorithm to identify regularity in the trace. The regularities are actually repeatedly executed sections of code. Each identified regularity is then compressed to form a unique element in the trace.

More specifically, the whole program path is constructed by collecting a control flow trace of the sequence of basic blocks executed by the program on a given input. To collect the trace we instrument the CFG by uniquely labeling each edge. As the program executes the edge labels are recorded, producing a trace. For example, consider the trivial program in Figure 12.1. At (a) we construct a CFG with six basic blocks and eight edges. During instrumentation each edge is uniquely labeled 1 through 8. Then at (b) the instrumented program is executed with the secret input sequence $I$ (which in this trivial example is a null input sequence) to produce the edge trace.

To reveal the inherent regularity and compress the trace, the trace is run through the SEQUITUR algorithm. The output of this step is a context-free grammar in which every production rule is of the form $R \rightarrow r$ where $R$ is a non-terminal symbol and $r$ is a string of terminal and/or non-terminal symbols. Continuing with the example in Figure 12.1 the execution trace is run through the SEQUITUR algorithm at (c) to produce a context-free grammar with three unique non-terminal and eight unique terminal symbols.

Based on the context-free grammar we construct a directed acyclic graph (DAG). For each symbol in the grammar a node is added to the DAG. The nodes are then connected by following the production rules. Given a production rule $R \rightarrow r$, a directed edge is added from $R$ to each symbol in the string $r$. If a symbol is repeated in the string, multiple edges are added from $R$ to the symbol. In Figure 12.1, at (d) a DAG is constructed with three internal nodes, eight leaf nodes, and 14 directed edges. This DAG is the whole program path representation of our trivial example program.
int a;
for (int i = 1; i < 5; i++) {
    if (i < 3)
        a = 1;
    else
        a = 2;
}

Figure 12.1: An illustration of the stages involved in constructing a WPP birthmark. The construction begins with a program. A program control flow graph is constructed and instrumented. By executing the program on a particular input, an edge trace is constructed. This trace is run through the SEQUITUR algorithm to produce a context-free grammar. The grammar is then used to construct a directed acyclic graph which represents the whole program path. Finally, the DAG is pruned to form the WPP birthmark.
An essential property of a birthmark is that it should capture an inherent characteristic of the program which is difficult to modify through semantics-preserving transformations. Since the whole program path captures the inherent regularity of the dynamic behavior of a program it ought to be an excellent basis for constructing a birthmark. Since we are only interested in the regularity we eliminate all terminal nodes in the DAG. It is the internal nodes which will be more difficult to modify through program transformations, thus, the DAG in Figure 12.1 is transformed at @ to form the WPP birthmark of the example program.

12.1.1 Similarity of WPP Birthmarks

The WPP birthmark is in the form of a DAG. In order to declare that a program $q$ is an identical copy of another program $p$ through the use of the birthmarks $b_p = G_p$ and $b_q = G_q$ we would have to show that $G_p$ and $G_q$ are isomorphic. Since it is unlikely that $q$ is an identical copy of $p$ we would like to be able to say something about the similarity between $b_p$ and $b_q$. In other words, we would like to be able to conclude that $q$ is a derivative of $p$ even in the presence of semantics-preserving transformations.

To compute similarity we use a slightly modified version of the graph distance metric proposed by Bunke and Shearer [25]. The similarity is based on finding a maximal common subgraph (MCS), $G_{pq}$, between $G_p$ and $G_q$. The percentage of $G_p$ that we are able to identify in $G_q$ by finding the MCS $G_{pq}$ indicates the similarity between the two programs. The reason we are comparing the size of $G_{pq}$ and $G_p$ instead of the maximum of $G_p$ and $G_q$ is that we are trying to identify a copy of $p$ in $q$. Therefore we want to know how much of $G_p$ is contained in $G_q$.

**Definition 21** (Graph Distance). The distance of two non-empty graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is defined as

$$d(G_1, G_2) = \frac{|\text{mcs}(G_1, G_2)|}{|G_1|}$$

where $\text{mcs}(G_1, G_2)$ is the maximum common subgraph of $G_1, G_2$ and $|G| = |V| + |E|$.
**Definition 22** (WPP Similarity). Let $b_p = G_p$ and $b_q = G_q$ be WPP birthmarks extracted from the programs $p$ and $q$ when executed with the input sequence $I$. The similarity between $b_p$ and $b_q$ is defined by:

$$\text{similarity}(b_p, b_q) = \frac{|mcs(G_p, G_q)|}{|G_p|}$$

### 12.1.2 Example

To help illustrate the WPP birthmarking technique we present a simple example in which we compute the birthmarks of an original and obfuscated version of a recursive factorial program. The obfuscation applied is a control flow obfuscation which makes a copy of a basic block of code and introduces a bug in it. The buggy code is used to confuse an attacker, but is never actually executed.

To produce a birthmark of the factorial program we start with the bytecode in Figure 12.2. From this bytecode a trace of the application is produced by annotating the edges of the program control flow graph and then executing the program. At \( \mathbb{1} \) the trace is run through the SEQUITUR program which produces the context-free grammar. Since the WPP birthmark is only focused on the non-terminal symbols in the grammar and DAG, a sequence of one or more terminal symbols is replaced in the grammar by [x terminals]. At \( \mathbb{2} \) the DAG representation of the grammar is constructed. For each rule an edge is added only if it connects a non-terminal to another non-terminal. This is the WPP birthmark. By repeating this procedure on the obfuscated program the grammar and DAG in Figure 12.3 are produced. What we can see from these two figures is that the size of our obfuscated birthmark is larger than the original birthmark.

To compute the similarity between the two birthmarks we compute the largest subgraph of the original DAG in the obfuscated DAG. Figure 12.4 illustrates that the WPP birthmarking technique identifies these two programs as 100% similar since the original DAG is completely contained in the obfuscated DAG.
Method `void main(java.lang.String[])`

```
0   getstatic #13 <Field java.io.PrintStream out>
6   dup
7   ldc #18 <String "15! = ">
9   invokevirtual #23 <Method java.lang.StringBuffer(java.lang.String)>
12  ldc2_w #24 <Long 15>
15  invokevirtual #29 <Method long fact(long)>
18  invokevirtual #33 <Method java.lang.StringBuffer.append(long)>
21  invokevirtual #37 <Method java.lang.String toString()>
24  invokevirtual #40 <Method void printIn(java.lang.String)>
27  return
```

Method `long fact(long)`

```
0   iload_0
1   lconst_0
2   lcmp
3   ifeq 12
6   iload_0
7   lconst_1
8   lcmp
9   ifne 14
12  lconst_1
13  ireturn
14  iload_0
15  iload_0
16  lconst_0
17  lsub
18  invokevirtual #29 <Method long fact(long)>
21  lmul
22  ireturn
```

Method `Main()`

```
0   aload_0
1   invokevirtual #49 <Method java.lang.Object()> 
4   return
```

Figure 12.2: WPP birthmark of a recursive factorial program.
12.1.3 Computing the Maximum Common Subgraph

The problem of finding the maximum common subgraph of two general graphs is known to be NP-hard [46]. However, when constrained by geometric information we can develop a heuristic. The WPP birthmark is represented as a DAG. Because the DAG is based on a context-free representation of an execution trace we know that it has a single root vertex. This provides a starting point for creating a subgraph isomorphism mapping between the two set of vertices. Additionally, since the graph is a DAG a topological sort of the vertices can be constructed. The topological sort is the main aspect which makes it possible for us to construct a mapping between $G_p$ and $G_q$.

The first step in computing the similarity is to construct a topological sort of the vertices for each of the graphs, $V_p = \{v_{p1}, \ldots, v_{pm}\}$ and $V_q = \{v_{q1}, \ldots, v_{qn}\}$. Because $v_{p1}$ and $v_{q1}$ are both root vertices in their respective graphs, we map $v_{p1}$ to $v_{q1}$. The mapping continues by examining the next vertex in each topological sort.
Figure 12.4: Using WPP birthmarking technique we can conclude that the original and the obfuscated programs are 100% similar since the original DAG is a subgraph of the obfuscated DAG.

If a match (see below for the definition of match) is made, the vertex pointers in each sort are incremented. If the current vertices do not match the vertex pointer for $V_q$ is incremented until a match is made or the end of the list is reached. After reaching the last vertex in $V_q$ it is known that the current vertex in $V_p$ does not have a corresponding vertex in $G_p$. Thus, the vertex pointer in $V_p$ is incremented and the vertex pointer for $V_q$ is reset to its most recent starting point.

A vertex in $V_q$ is considered a match for a vertex in $V_p$ if all of the following conditions are met:

1. $\text{indegree}(v_{p_i}) \leq \text{indegree}(v_{q_i})$.
2. $\text{outdegree}(v_{p_i}) \leq \text{outdegree}(v_{q_i})$.
3. The number of predecessors of $v_{p_i} \leq$ the number of predecessors of $v_{q_i}$.
4. All predecessors of $v_{p_i}$ map to a predecessor of $v_{q_i}$ or to no vertex in $G_q$.
5. No predecessor of the $v_{q_i}$ is mapped to a non-predecessor of $v_{p_i}$.
Condition 3 is necessary because two vertices can be connected by multiple edges, thus the indegree of a vertex does not reflect the number of predecessor vertices. Condition 5 is only necessary in instances where a predecessor of \( v_{pi} \) does not have a corresponding vertex in \( G_q \). When this occurs verification of the predecessors of \( v_q \) is necessary.

Unfortunately as is, this idea is not sufficient to identify the MCS for all graphs which we could encounter. For example consider the two graphs in figure 12.5. Graph (a) is the WPP birthmark for the iterative factorial program in Appendix D and (b) is the WPP birthmark of the same program except the Smokescreen obfuscation has been applied. Using the above idea, the MCS would only have two vertices. It would be created by mapping \( R0_p \) to \( R0_q \), \( R1_p \) to \( R1_q \).

To address this suboptimal solution we draw on what we call second roots. A second root is any node whose only predecessor is the actual root. For each second root in \( G_q \) we repeat the mapping procedure. We then use the mapping which consists of the greatest number of nodes.

In summary, the algorithm used by the WPP birthmarking system to compute the maximum common subgraph is as follows:
Algorithm mcs($G_p$, $G_q$):
1. $V_p$ ← toposort($G_p$);
2. $V_q$ ← toposort($G_q$);
3. finalMapping ← new Hashtable;
4. secondRoots ← secondRoots($G_q$);
5. for $i$ ← 0; $i$ < secondRoots; $i$ ++ do
6.   vpPointer ← 1;
7.   vqPointer ← $i$ + 1;
8.   vertexMapping ← new Hashtable;
9.   proot ← $V_p[0]$; qroot ← $V_q[0]$;
10. vertexMapping.put(proot, qroot);
11. while vpPointer < $V_p$.length and vqPointer < $V_q$.length do
12.   $p$ ← $V_p[vpPointer]$; $q$ ← $V_q[vqPointer]$;
13.   if foundMatch($p$, $q$) then
14.     vertexMapping.put($p$, $q$);
15.     vpPointer ++;
16.     vqPointer ++;
17.   else
18.     tmpvqPointer ← vqPointer + 1;
19.     while tmpvqPointer < $V_q$.length do
20.        tmpq ← $V_q[tmpvqPointer]$;
21.        if foundMatch($p$, tmpq) then
22.           vertexMapping.put($p$, tmpq);
23.           tmpvqPointer ← $V_q$.length;
24.        else
25.           tmpvqPointer ++;
26.           vpPointer ++;
27.        end if
28.     end while
29.   end else
30. end if
31. if vertexMapping.numElements > finalMapping.numElements do
32.   finalMapping ← vertexMapping;
33. end if
34. return finalMapping;
Using this \texttt{mos} algorithm we can analyze the cost associated with measuring the similarity between two \textit{WPP} birthmarks. The first step in the algorithm is computing the topological sort for the two DAGs. The running time for this step is $O(|V| + |E|)$ for each graph. The main body of the algorithm has a triple nested loop. The outer loop for at line 5 is based on the number of second roots in $G_q$. Because $G_q$ has exactly one root we know $|secondRoots| \leq |V_q| - 1$. The next loop level occurs at line 11. This while is dependent on two conditions and in the worst case may execute $O(|V_p| \times |V_q|)$ times. Finally, we have the inner most loop which could execute $O(|V_p|)$ times. This gives us an overall running time of $O(|V_p| \times |V_q|^2)$. In addition to the running time there is also a space concern. In our implementation the DAG is stored in an adjacency matrix and thus the space requirement is $O(|V_p|^2 + |V_q|^2)$. Table 12.1 shows the number of nodes and edges for the graphs in our simple testcases. From this it should be clear that the complexity of computing the similarity is a major limiting factor in using this technique to detect theft of large programs.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
 & IterFact & IterFib & RecurFact & RecurFib & decode & fft & wc \\
\hline
Original & 4, 9 & 4, 7 & 6, 14 & 23, 57 & 649, 2533 & 345, 1032 & 151, 619 \\
\textit{SandMark} Test 1 & 4, 9 & 4, 7 & 7, 14 & 23, 72 & 649, 2533 & 370, 1116 & 151, 619 \\
\textit{SandMark} Test 2 & 4, 9 & 4, 7 & 8, 18 & 25, 61 & 649, 2533 & 345, 1032 & 151, 619 \\
\textit{SandMark} Test 2 & 32, 88 & 36, 105 & 36, 97 & 57, 159 & 556, 2920 & 458, 1473 & 159, 639 \\
\textit{SandMark} Test 3 & 5, 11 & 5, 9 & 8, 18 & 26, 65 & 673, 2688 & 366, 1068 & 153, 627 \\
\textit{SandMark} Test 4 & 5, 11 & 4, 7 & 6, 14 & 27, 76 & 597, 2388 & 347, 1037 & 151, 619 \\
\textit{SandMark} Test 5 & 5, 11 & 5, 9 & 9, 20 & 24, 61 & 649, 2533 & 362, 1068 & 151, 619 \\
\textit{SandMark} Test 6 & 5, 11 & 5, 9 & 7, 16 & 26, 65 & 673, 2688 & 336, 1068 & 153, 627 \\
\textit{Simulcast} & 7, 16 & 8, 15 & 9, 21 & 27, 65 & 606, 2831 & 350, 1076 & 155, 632 \\
\textit{CodeShelf} & 4, 9 & 4, 7 & 7, 16 & 24, 59 & 649, 2533 & 345, 1032 & 151, 619 \\
\textit{ZeEx Klassmaster} & 7, 18 & 8, 18 & 10, 25 & 29, 72 & 672, 2606 & 367, 1111 & 155, 633 \\
\textit{jarg} & 4, 9 & 4, 7 & 6, 14 & 23, 57 & 649, 2533 & 335, 1029 & 151, 619 \\
\hline
\end{tabular}
\caption{The number of nodes and edges in each of the original and obfuscated applications used to evaluate the \textit{WPP} birthmarking technique.}
\end{table}
12.2 Evaluation

To evaluate the effectiveness of the WPP birthmarking technique we examined its ability to satisfy the properties of credibility and resistance to transformation which were defined in Chapter 10. The evaluation was conducted using the framework described in Section 10.3. To compute similarity using the WPP birthmarking technique the test programs have to be executed. Two of the seven applications require user input to execute: decode and wc. decode requires two input parameters: the number of shares and a cipher text. We chose 10 as the number of shares. For wc we used the first chapter of the Count of Monte Cristo by Alexander Dumas as the input data. This chapter contains 488 lines, 3165 words, and 17439 bytes. To demonstrate the strength of the WPP birthmarking technique we compare the evaluation results to those of the two static techniques from the previous chapter. As an additional evaluation we demonstrate how birthmarks can be used in conjunction with watermarking.

12.2.1 Credibility

To evaluate the credibility of the WPP birthmarking technique we examined the possibility of producing faulty results using the two credibility experiments. In the independence experiment the ability to distinguish between independently implemented applications which perform the same task is demonstrated through the factorial and Fibonacci programs. Table 12.2 shows that the programs exhibit some similarity. From these results we cannot completely rule out an incidence of theft, however, the results are low enough to conclude that the programs do have significant differences and may have been implemented independently. (For the exact similarity results for the WPP birthmarking technique see Appendix E.)

In the false positive experiment the similarity was computed between 19 pairings of seven Java applications. From Table 12.3 it can be seen that the WPP birthmarking technique demonstrates a relatively high level of credibility on the test data. In all except four cases the pairs were accurately flagged as not derivatives. Of these
<table>
<thead>
<tr>
<th>Program</th>
<th>WPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>±</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>±</td>
</tr>
</tbody>
</table>

⊕ : derivative  
± : inconclusive  
⊖ : not derivative

Table 12.2: Results for the independence experiment using the WPP birthmarking technique.

four only one was incorrectly reported as derivative and this probably occurred because both programs are small with the majority of the execution occurring within a single for loop.

12.2.2 Resistance to Transformation

To evaluate the WPP birthmarking technique’s resistance to transformation we applied the ten obfuscation/optimization tests to automatically transform our test programs into equivalent, but not identical programs.

We do know of two attacks to which the WPP birthmarking technique is currently vulnerable. The first is any loop transformation that alters the loop in ways similar to loop unrolling or loop splitting. Executing the loop backwards, however, will not affect the WPP birthmark. WPP birthmarks are also vulnerable to method inlining in certain instances. If the method call occurs inside of a loop then inlining will not alter the birthmark. On the other hand, if the method is a helper method which is called from various locations throughout the program, inlining the method call will have an effect on the birthmark similarity. SandMark Test 1 and Test 4 both include obfuscations which perform inlining. Table 12.4 shows that these two obfuscations are able to damage the WPP birthmark. To demonstrate that the birthmark is resistant to the other transformations in Test 1 and Test 4 we
<table>
<thead>
<tr>
<th>Programs</th>
<th>WPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IterFact, IterFib</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFact, RecurFib</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFact, decode</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFact, fft</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFact, wc</td>
<td>⊕</td>
</tr>
<tr>
<td>RecurFact, IterFib</td>
<td>±</td>
</tr>
<tr>
<td>RecurFact, RecurFib</td>
<td>⊕</td>
</tr>
<tr>
<td>RecurFact, decode</td>
<td>⊕</td>
</tr>
<tr>
<td>RecurFact, fft</td>
<td>⊕</td>
</tr>
<tr>
<td>RecurFact, wc</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFib, decode</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFib, fft</td>
<td>±</td>
</tr>
<tr>
<td>IterFib, wc</td>
<td>⊕</td>
</tr>
<tr>
<td>RecurFib, decode</td>
<td>⊕</td>
</tr>
<tr>
<td>RecurFib, fft</td>
<td>⊕</td>
</tr>
<tr>
<td>RecurFib, wc</td>
<td>⊕</td>
</tr>
<tr>
<td>decode, fft</td>
<td>⊕</td>
</tr>
<tr>
<td>decode, wc</td>
<td>⊕</td>
</tr>
<tr>
<td>fft, wc</td>
<td>⊕</td>
</tr>
</tbody>
</table>

⊕ : derivative  
± : inconclusive  
⊕ : not derivative

Table 12.3: Results for false positive experiment using the WPP birthmark technique.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>IterFact</th>
<th>IterFlib</th>
<th>RecurFact</th>
<th>RecurFlib</th>
<th>decode</th>
<th>fft</th>
<th>wc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM Test 1</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SM Test 1a</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SM Test 2</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SM Test 3</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SM Test 4</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SM Test 4a</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SM Test 5</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>SM Test 6</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>Codenick</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>jarg</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
</tbody>
</table>

⊕ : derivative
± : inconclusive
⊖ : not derivative

Table 12.4: Results from the resistance experiment using the WPP birthmarking technique.

constructed SandMark Test 1a and SandMark Test 4a which include all of the obfuscations in Test 1 and Test 4 except inlining. As part of our future work we would like to explore techniques that would make the WPP technique more resistant to loop transformations. For example, through the addition of data dependencies it is possible to prevent loop unrolling.

Table 12.4 also shows that for a single testcase each, Zelix Klassmaster and jarg are able to defeat the WPP birthmarking technique. The overall resistance results in Table 12.4 demonstrates the WPP birthmarking technique is highly resistant to transformation except when inlining is used.

12.2.3 Technique Comparison

In order to provide a full analysis of the WPP birthmarking technique we draw on the evaluation of the two static birthmarking techniques from the previous chapter. Table 12.5 shows a comparison of the results from the independence experiment. From this we can see that the WPP technique is better at distinguishing between
<table>
<thead>
<tr>
<th>Program</th>
<th>TaNaMM</th>
<th>K-Gram</th>
<th>WPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>⊕</td>
<td>⊗</td>
<td>±</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>⊕</td>
<td>⊗</td>
<td>±</td>
</tr>
</tbody>
</table>

⊕ : derivative  
± : inconclusive  
⊗ : not derivative

Table 12.5: Results from the independence experiment comparing all three birthmarking techniques.

small independently written applications which accomplish the same task than the TaNaMM technique. However, the results are not as good as when the K-Gram technique is used.

The overall credibility of a birthmarking system is also evaluated using the false positive experiment. From Table 12.6 it is clear that the WPP technique provides fewer false positives than the TaNaMM technique. The WPP technique only incorrectly flags one pair as derivative as opposed to the 16 pairs by the TaNaMM technique. Similar to the independence experiment, the WPP technique provides slightly weaker results than the K-Gram technique with respect to the false positive rate. Both techniques incorrectly flag a single pairing as derivatives, however, the WPP technique also reports three inconclusive results.

For the two static birthmarking techniques, our analysis in the previous chapter showed that the TaNaMM technique demonstrates a higher level of resistance to transformation. For every test application and obfuscation/optimization, the TaNaMM technique accurately reported derivative. The WPP technique does not achieve quite as high results, however the results are higher than the K-Gram technique. The WPP technique is vulnerable to transformations such as loop unrolling and method inlining to which the TaNaMM technique is resistant. Table 12.7 provides a comparison between the resistance results for each of the birthmarking tech-
<table>
<thead>
<tr>
<th>Programs</th>
<th>TaNaMM</th>
<th>K-Gram</th>
<th>WPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IterFact, IterFib</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFact, RecurFib</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFact, decode</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFact, ft</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>IterFact, wc</td>
<td>⊕</td>
<td>⊕</td>
<td>⊕</td>
</tr>
<tr>
<td>RecurFact, IterFib</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>RecurFact, RecurFib</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>RecurFact, decode</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>RecurFact, ft</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>RecurFact, wc</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>IterFib, decode</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>IterFib, ft</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>IterFib, wc</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>RecurFib, decode</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>RecurFib, ft</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>RecurFib, wc</td>
<td>⊕</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>decode, ft</td>
<td>±</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>decode, wc</td>
<td>±</td>
<td>⊕</td>
<td>±</td>
</tr>
<tr>
<td>ft, wc</td>
<td>±</td>
<td>⊕</td>
<td>±</td>
</tr>
</tbody>
</table>

⊕ : derivative

± : inconclusive

⊕ : not derivative

Table 12.6: Results from the false positive experiment comparing all three birthmarking techniques.

Overall, these results lead us to believe that it may be quite difficult to achieve the highest level of both credibility and resistance to transformation. The TaNaMM technique has a very high level of resistance to transformation but a low degree of credibility. The K-Gram technique is the reverse: a fairly high degree of credibility but only average resistance to transformation. Of the three techniques, the WPP technique provides the best balance between the two properties. It demonstrates moderately high levels of both credibility and resistance to transformation.

### 12.2.4 Birthmarks and Watermarks

One limitation of software birthmarking is that it provides weaker evidence than software watermarking. It is only able to say that one program is likely to be a
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>IterFact</th>
<th>RecurFact</th>
<th>RecurFib</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TaNaMM</td>
<td>K-Gram</td>
<td>WPP</td>
</tr>
<tr>
<td>SANDMark Test 1</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SANDMark Test 1a</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SANDMark Test 4a</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>✪</td>
<td>±</td>
<td>✪</td>
</tr>
<tr>
<td>Codeshield</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>✪</td>
<td>±</td>
<td>✪</td>
</tr>
<tr>
<td>jarg</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>decode</th>
<th>fit</th>
<th>we</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TaNaMM</td>
<td>K-Gram</td>
<td>WPP</td>
</tr>
<tr>
<td>SM Test 1</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SM Test 1a</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SM Test 2</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SM Test 3</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SM Test 4</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SM Test 4a</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SM Test 5</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>SM Test 6</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>Codeshield</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>Klassmaster</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
<tr>
<td>jarg</td>
<td>✪</td>
<td>✪</td>
<td>✪</td>
</tr>
</tbody>
</table>

⊕ : derivative  
± : inconclusive  
⊖ : not derivative

Table 12.7: Results from the resistance experiment using all three birthmarking techniques.
copy of another, not who the original author is or who is guilty of piracy. However, birthmarking can be used in instances where watermarking is not feasible such as applications where code size is a concern and the watermark would insert additional code. Additionally, since the watermarking process transforms the code it is often considered too dangerous by software quality assurance people. Finally, software birthmarking could be useful in legal disputes when watermarking was not used. For example, if the program in question was developed prior to the advent of software watermarking, it will not contain a watermark but we can still extract a birthmark.

Birthmarking can also be used in conjunction with watermarking to provide stronger evidence of theft. One such example is the watermarking algorithm proposed by Stern, et. al. [93] which provides a probability that a specific watermark is contained in the program. If the watermarking algorithm does not 100% guarantee that the watermark is contained in the program then birthmarking could be used as additional evidence of theft. There are also instances where watermarks fail, e.g. when an attacker is able to apply an obfuscation which destroys the watermark. In these instances birthmarking may still be able to provide proof of program theft.

We were able to very easily construct two instances where a watermark is destroyed by an obfuscation and the WPP birthmarking technique still detects 100% similarity between the programs. For the first example, we used the Spread Spectrum watermarking algorithm which was presented in Chapter 7 and the wc application. In this experiment we watermarked the application and then applied the buggy code obfuscation. For our second experiment we watermarked the wc program using the Arboit algorithm presented in Chapter 7. We then applied the opaque branch insertion obfuscation In each of these instance the watermark is no longer detectable which would have prevented piracy detection, but the WPP birthmarking technique was able to detect a derivative.
12.3 Summary

In this chapter we described a novel approach to software birthmarking, WPP birthmarking, which is based on the whole program path idea by Larus to represent a program’s dynamic control flow in a compressed form. Our technique is an improvement over previous techniques because it provides more of a balance between the properties of credibility and resistance to transformation. The current implementation of the WPP birthmarking technique is vulnerable to various loop transformations, however in future work we plan to explore techniques such as loop rerolling to account for these transformations. Additionally, extensive time and space is required to construct the birthmarks and compute the similarity. These are both issues we would like to explore in the future. Overall the WPP birthmarking technique makes several contributions over the techniques presented in the previous chapter:

1. It is the first known dynamic birthmarking technique.

2. The technique demonstrates a stronger balance between the properties of credibility and resistance to transformation.

3. The high degree of resistance to transformation makes it possible to identify program theft in instances when an embedded watermark has been destroyed.
CHAPTER 13

CONCLUSION

Currently there are three major threats recognized against the intellectual property in software: malicious reverse engineering, software tampering, and software piracy. In this research we focused on addressing the threat of software piracy. Software piracy is a widespread, decentralized problem which has been compounded several factors. These include rich distribution formats, such as Java and .Net; the availability of high speed Internet and peer-to-peer systems; and the difficulty of detecting illegal copies. In many countries legal methods exist to combat software piracy. Unfortunately, due the difficulty in detecting an incidence of theft, these laws, which rely on a fear of consequences, offer limited effectiveness. In a response to this limitation a variety of technical approaches are being explored to prevent and detect individual cases of piracy.

The research in this dissertation focused on two different techniques which make it possible to detect individual occurrences of software theft. The first technique we explored is that of software watermarking. In software watermarking a unique identifier is embedded in the program. Piracy is confirmed by proving the program contains the watermark. Depending on the type of mark embedded, watermarking is used to prove authorship or to trace the source of an illegal redistribution.

In our research we analyze the state of the art in software watermarking. This analysis includes an extensive discussion, evaluation, and comparison of the most comprehensive collection of published software watermarking algorithms to date. It is our hope that this analysis will serve as a basis for other researchers studying the area of software watermarking.

In addition to the extensive analysis of published watermarking algorithms, we proposed a novel, dynamic software watermarking technique: the Branch-Based algorithm. This technique was designed to improve on the common vulnerabilities
of previous algorithms by incorporating ideas from code obfuscation and software
tamper detection. The main idea is to incorporate a specifically designed branch
function which will generate the fingerprint as the program executes. This basic
idea can be implemented in a variety of different ways. Many of the implementation
details are architecture specific; however, variations can be constructed based on the
architectural characteristics.

The Branch-Based algorithm makes several important contributions to the area
of software watermarking. First, it is able to simultaneously provide proof of own-
ership and trace the source of an illegal distribution. This an improvement over
previous techniques which required the developer to choose between the protection
techniques. Second, because the only static variation between two differently fin-
gerprinted copies occurs in the inserted table, the Branch-Based algorithm can be
used to distribute pre-packaged, fingerprinted software which is linked to the con-
sumer. Finally, the algorithm demonstrates a significantly higher level of resilience
to attack without significant overhead. The Branch-Based algorithm can successfully
thwart both additive and distortive attacks and is highly resistant to subtractive and
collusive attacks.

The second technique we explored is that of software birthmarking. Software
birthmarking is used to address the illegal distribution of all or some part of a pro-
gram by extracting identifying characteristics from the programs. These characteris-
tics are then compared to show that one program is derivative of the other. Software
birthmarking differs from software watermarking in two important ways. First, in
order to embed a watermark it is often necessary to add code to the application.
Additional code is never needed for software birthmarking. Second, birthmarking
cannot prove authorship or be used to identify the source of an illegal redistribution.
Instead, birthmarking simply confirms that one program is a copy of another. How-
ever, a strong birthmarking system will be able to provide such confirmation even
when the adversary has applied semantics-preserving transformations in an attempt
to disguise the incidence of theft.

In the area of software birthmarking our research makes two important con-
tributions. First, we extend the definition of software birthmarking by proposing the idea of a dynamic software birthmarking technique. Second, we propose the first dynamic birthmarking technique: the **Whole Program Path** birthmark. This technique is language independent and constructs the birthmark by capturing a representation of the program's dynamic control flow as it executes on a particular input. **WPP** birthmarking is an improvement over previous techniques because it provides a stronger balance between the properties of credibility and resistance to transformation. Additionally, the high degree of resistance to transformation makes it possible to identify program theft in instances when an embedded watermark has been destroyed.

### 13.1 Future Work

In this research we presented two different approaches to address the threat of software piracy. In both the areas of software watermarking and software birthmarking we proposed a technique which makes significant improvements over the previous techniques. Despite this fact, improvements and further research is still required in each area.

Over the past several years research in the area of software watermarking has been quite active. The number and sophistication of the algorithms has greatly increased. However, based on the thorough evaluation presented in Chapter 7 it should be clear that a perfect watermarking algorithm has yet to be developed. The **Branch-Based** algorithm does make several improvements over previous algorithms due to its resistance to both distortive and additive attacks, but unfortunately the algorithm can be defeated by a truly determined adversary.

In addition to research on developing stronger watermarking algorithms, a meaningful way to measure the stealth of an algorithm has to be developed. Currently, the level of stealth exhibited by an algorithm is a purely subjective measure. In this research we categorized stealth as either statistical or visual. To evaluate the statistical stealth of an algorithm it will be necessary to devise a mathematical measure.
However, to develop a measure of visual stealth will probably require extensive user studies. Once a technique is developed for measuring both types of stealth the role stealth has in the overall strength of a watermarking technique should be investigated.

Research into software birthmarking techniques and how they can be used to combat software theft is still in its infancy. The three birthmarking techniques presented in this research demonstrate that we have yet to develop a truly useable birthmarking scheme. The WPP algorithm demonstrates the best balance between credibility and resistance to transformation but is costly to compute for real programs. It is our plan to make further improvements to the WPP technique by making it more resistant to loop transformations. However, the technique is ultimately limited by the complexity of computing the similarity. To address this limitation new birthmarking techniques should be developed which also demonstrate a high degree of credibility and resistance to transformation.

Overall, in addressing the threat of software piracy there is still significant work to be done. Currently there is no single mechanism which makes it possible to completely prevent the threat. Software watermarking and software birthmarking are just two of the possible techniques which can be used to address software theft. However, neither of these techniques are able to prevent the attack. Instead they are used after the fact to prove that an incidence of piracy has occurred. To fully address the issue of piracy we must develop techniques which not only allow us to identify and trace the theft, but also make it more difficult, and ultimately impossible, for an attacker to make viable copies and redistribute them. In Chapter 2 we presented an overview of both hardware- and software-based approaches to software protection. A truly strong technique which prevents software piracy may require the development of a technique which leverages both hardware- and software-based protection mechanisms.
APPENDIX A

ASSEMBLY LEVEL PROGRAMMING: JAVA AND x86

The software watermarking and birthmarking algorithms presented in the following chapters manipulate and/or analyze a program at the assembly level. In particular, the watermarking algorithms make use of assembly level constructs and processor specific details in order to embed the watermark in the application. The majority of the research has been implemented within the SANDMARK framework and thus operates at the Java bytecode level. However, the branch based software watermarking algorithm presented in Chapter 9 was initially developed to protect native executables. It therefore makes use of features specific to the x86 instruction set and the x86 line of CPUs. In this chapter we present an overview of Java bytecode, the Java Virtual Machine, and x86 assembly language programming to provide a foundation for discussing the details of the various software protection algorithms.

A.1 Java Bytecode and the Java Virtual Machine

The Java technology is comprised of three distinct parts: the Java programming language, the compiled Java language called Java bytecode, and the Java Virtual Machine (JVM). To run a program in the simulated environment of the JVM the source code must be compiled to Java bytecode. It is common that the source code is writing using the Java programming language, however, it is possible to compile programs written in other languages to Java bytecode. Additionally, Java source code can be compiled to native code and run on a hardware CPU. One important feature of the Java technology is the ability to ‘compile once, run everywhere.’ When the program is compiled to Java bytecode it is platform independent. This portability is lost when the Java source code is compiled to native code.

In this research the watermarking and birthmarking techniques are performed at
the Java bytecode level. We are not interested in the original source code. Instead, we are interested in the ways that Java bytecode can be manipulated to both incorporate protection mechanisms and circumvent those mechanisms. This requires an understanding of Java bytecode applications as well as the JVM. The overview of these two parts of the Java technology is limited to those aspects relevant to our theft detection techniques. A more detailed treatment of these topics can be found in *The Java Virtual Machine Specification* [63].

### A.1.1 The Class File Format

A Java source file can contain several Java classes. When the source file is run through a Java compiler each Java class is compiled to a single class file, which contains all of the details necessary to execute the class, on a compliant JVM, in a strict format. These details include the bytecode instructions and all identifier information from the source code. The structure of the class file makes it possible to provide platform independence as well as verification that the application is not trying to harm the host machine. The structure of the class file is as follows:

- **magic** - a number identifying the class file format.
- **minor_version, major_version** - identifies the version of the class file format.
- **constant_pool_count** - the number of constant pool entries plus 1.
- **constant_pool[]** - an array of typed values representing various constants in the program such as class and method names.
- **access_flags** - flags to indicate access permissions of the class or interface.
- **this_class** - an index into the constant pool table.
- **super_class** - an index into the constant pool table.
- **interfaces_count** - the number of direct superinterfaces of the class or interface.
• **interfaces[]** - an array of indices into the constant pool table where a structure representing each superinterface can be found.

• **fields_count** - the number of fields in the class or interface.

• **fields[]** - an array of **field_info** structures which provide complete descriptions of the fields in the class or interface.

• **methods_count** - the number of methods in the class or interface.

• **methods[]** - an array of **method_info** structures which provide complete descriptions of the methods in the class or interface, including the bytecode instructions.

• **attribute_count** - the number of attributes in the class or interface.

• **attributes[]** - an array of attribute structures.

The execution of a Java class file heavily relies on the symbolic information stored in the **constant_pool** table. For example, an instruction which references a field does so by specifying the **constant_pool** index of that field. The **constant_pool** table provides a level of detail not found in most compiled programs. The strict class file format and level of detail aid in the verification process, but they also make it a straightforward task to modify the application or recover the source code. Both have implications on protecting the program from piracy.

A.1.2 The JVM

The JVM maintains several runtime data areas which are used as the program executes. The runtime data areas include the method area, JVM stack, pc register, and native method stack. Each instance of the JVM has one method area and one heap. They are created when the JVM starts up and remains until the JVM exits. Each thread running inside the JVM share these two data areas. The remaining three data areas are created specifically for each individual thread. These data areas are created when the thread is created and destroyed on thread exit.
The method area stores execution information on a per class basis. For each class the method area maintains the runtime constant pool, field and method data, and the instruction for the methods. The memory used for the method area may expand and contract or remain fixed throughout JVM execution. The method area is actually part of the heap. In addition to providing memory for the method area, the heap is the area from which all class instances and arrays are allocated.

For each thread a private JVM stack is created. Upon method invocation a new frame is pushed onto the stack. The purpose of the frame is to store method data, maintain partial results, perform dynamic linking, and dispatch exceptions. Each frame maintains an array of local variables, an operand stack, and a reference to the runtime constant pool. The runtime constant pool is used to support dynamic linking. The symbolic references stored in the constant pool are translated to concrete references. Any undefined symbols are resolved by loading the appropriate classes. Once the current method completes execution any results are passed to the previous frame and the current frame is popped from the stack.

As code executes on an individual thread, the current execution location is maintained by the program counter (pc) register. The pc register contains the address of the instruction currently being executed. During certain instances the pc register may contain an address of type `returnAddress`. The `returnAddress` type is used in the implementation of the `try-finally` construct. The `finally` clause is implemented as a subroutine. The `jar` bytecode instruction is used to invoke the subroutine. The address of the next instruction is pushed onto the operand stack as a value of type `returnAddress`. The code for the subroutine stores the return address in a local variable. To complete the subroutine the `ret` instruction is executed which obtains the return address from the local variable and transfers control to the instruction at that address. Because the `returnAddress` type does not correspond to a Java programming language type, it cannot be modified by the running program. Additionally, there is no way to directly access and modify the pc register as the program executes.

The final runtime data area is the native method stack. The JVM supports the
execution of native methods, those methods written in a language other than Java. The native method stack is an implementation of the conventional stack used to execute native methods.

One of the important features of the JVM is the ability to protect the host machine from malicious programs. To accomplish this a Java program must pass the verification process prior to execution. Verification involves checking that the class file as well as the bytecode meet certain constraints at link time. By performing the check at link time many costly runtime checks can be avoided. The verification process is performed in four passes:

1. The structure of the class file is examined to ensure that it meets the basic format.

2. All additional verification that does not require looking at the code is performed. This includes checking that:
   - Final classes are not subclassed and final methods are not overridden.
   - Every class has a direct superclass.
   - The constant pool satisfies the static constraints.
   - All field and method references are well formed.

3. Data flow analysis is performed on each method to ensure that no matter what path is taken to reach a specific point, the following is true:
   - The operand stack is a consistent size and maintains the same value types.
   - Only local variables initialized to the proper type are accessed.
   - Methods are invoked with the correct arguments.
   - Fields are assigned values of the correct type.
   - All opcodes use the appropriate argument types.

4. All additional verification that requires loading class files is performed. This final pass is performed the first time the code is actually invoked. For example,
if an instance of type A is assigned to a field of type B, it must be ensured that A is a subclass of B.

A.1.3 Java Bytecode

An instruction in the Java bytecode instruction set consists of a one byte opcode, which specifies the operation to perform, followed by zero or more operands, which supply the arguments or data. Since the JVM is strongly typed most of the instructions encode type information. For example, the `istore` instruction stores the top item on the stack, which must be an `int`, to a local variable. The `lstore` instruction does the same with a `long`. The basic types used within the JVM include `int`, `long`, `float`, `double`, `byte`, `short`, and `Object` reference (pointer). These types are encoded in the instructions using the mnemonics `i`, `l`, `f`, `d`, `b`, `s`, and `a`. The instruction set consists of 201 usable opcodes which can be split into the following nine categories: load and store; arithmetic; type conversion; object creation and manipulation; stack management; control transfer; method invocation and return; exception throwing; and synchronization. Figure A.1 shows a Java method and its corresponding bytecode.

A.2 x86 Assembly Language Programming

In the previous section we described some of the details associated with a Java bytecode program and the virtual environment in which it is executed. An alternative is to compile the source code to a native executable and run the program directly on the system’s processor. Such a technique has many advantages such as improved performance and greater control over the execution environment as the program runs. Unfortunately, this requires the program to be compiled for a specific platform, limiting code portability. For example, a program compiled to execute on a Pentium processor running Unix will not execute on the same Pentium processor running Windows. This is due to the file formats used by the various operating systems. Additionally, a program compiled to execute on a Pentium processor will not
execute on a PowerPC processor due to architecture and instruction set variations. In this research we are particularly interested in the 32-bit x86 architecture and its associated instruction set.

The 32-bit x86 processors have 8 32-bit registers. Of these, four are accumulator registers:

- EAX – primary accumulator register; holds the return value from a function call.
- EBX – often used to hold the starting address of an array.
- ECX – often used as a counter for a loop or an index register for an array.
- EDX – general purpose register.

Two registers are reserved for special purposes. The EBP register is the stack frame pointer. It is used to aid in function calls and returns. The ESP register is the stack pointer. It always points to the top of the stack. The remaining two registers, EDI and ESI, are general purpose registers.
Figure A.2: A pictorial representation of the x86 stack for a single function.

To aid in program execution the x86 processor maintains a stack in memory. The stack is used to store local variables and data for making function calls. By convention, for each function call the function’s Activation Record is placed on the stack. The Activation Record consists of function parameters, the return address, the old frame pointer, and local variables. A pictorial representation of the stack in memory for a single function can be seen in Figure A.2. The stack can be manipulated through the push and pop instruction or the mov instruction to store data in the allocated space. During function execution, the frame pointer is used to reference the allocated memory associated with the stack. When the function exits any local variables are removed from the stack; the frame pointer is reset to the old value; and the return address, which is now at the top of the stack, is used to update the program counter. Once control is transferred back to the calling function, any parameters are popped from the stack.

The x86 instruction set uses a two address, variable length instruction. Each instruction can operate on a memory location, but the remaining operand must be a register. Thus, at most one memory location can be used as the operand of an instruction. The instruction set can be grouped into 3 general categories: data movement, arithmetic/logic, and control flow. Within the instruction four different addressing modes can be used:

- Register Mode: A register mode operand names a register.

- Immediate Mode: An immediate mode operand is a constant listed directly in the code.
void forFunction(){
    for(int i=0; i < 5; i++){
        printf("this_is_i_\%d\n", i);
    }
}

push ebp
mov ebp, esp
push ecx
mov dword ptr [ebp-04h], 0h
jmp L2
L1:
    mov eax, [ebp-04h]
    add eax, 01h
    mov [ebp-04h], eax
L2:
    cmp dword ptr [ebp-04h], 05h
    jge L3
    mov ecx, [ebp-04h]
push ecx
push string
call printf
add esp, 08h
jmp L1
L3:
    mov esp, ebp
pop ebp
ret

Figure A.3: A C function and the corresponding assembly instructions expressed in Intel notation.

- Register Indirect: A register indirect operand uses a register to hold an address in memory. Only one operand can be register indirect.

- Base Displacement: A base displacement operand allows for an offset from some base address.

Figure A.3 shows a C function and its corresponding assembly instructions expressed in Intel notation where the destination always comes before the source.
APPENDIX B

Description of SandMark Obfuscations

**Array Folder** - Takes a one-dimensional array and ‘folds’ it into a multi-dimensional array.

**Array Splitter** - Takes a one-dimensional array field and splits it into 2 arrays by adding another field of the same type. One array will contain the first half of the elements and the other will contain the second half.

**Block Marker** - Randomly marks all basic blocks in the program with either a 0 or 1.

**Bludgeon Signatures** - Converts all static methods to take Object[] as their parameter and return Object.

**Boolean Splitter** - Every boolean variable or array element is split into 2 variables or array elements, and the state of the original variable is reflected in the combined state of these 2 variables or array elements.

**Branch Inverter** - Negates an if instruction in the if-else statement and exchanges the if and else part of the body.

**Buggy Code** - Makes a copy of a basic block of code and introduces a bug in it. The buggy code is never executed but jumped over by means of an opaque predicate.

**Class Splitter** - A class C is split into two classes C1 and C2, such that C2 inherits from C1. C1 has fields and methods that only refer to themselves, whereas C2 has fields and methods that can refer to themselves as well as fields and methods in C1. References to C will be replaced with references to C2.
**Constant Pool Reorderer** - Randomly reassigns constant pool indices.

**Dublicate Registers** - Takes a local variable in a method and splits references to it with a new variable.

**Dynamic Inliner** - Inlines, non-static methods, determining which branch to use at runtime.

**Exception Branches** - Replaces all branches that have an empty stack (except the jsr instruction) with throw instructions.

**False Refactor** - Merges two classes C1 and C2 by adding a bogus parent class C3. If both classes have instance variables of the same type, these can be moved into C3.

**Field Assignment** - Adds a bogus field to each class in an application and throughout the class makes assignments to the field.

**Inliner** - Inlines static methods.

**Insert Opaque Predicates** - Appends an opaquely false opaque predicate, supplied by the opaque predicate library, to every boolean expression in a method.

**Interleave Methods** - Combines two methods into one new method. The execution path is based on the value of an opaque predicate input as an added parameter to the new method.

**Irreducibility** - Inserts jumps into a method via opaque predicates so that the control flow graph is irreducible.

**Merge Local Integers** - Combines two int variables into a single long variable.

**Method Merger** - Merges all of the public static methods that have the same signature in each class into one larger master method.

**Objectify** - Replaces all fields with fields of type Object.
Opaque Branch Insertion - Inserts an empty if block before a configurable fraction of the instructions in a method. The inserted test uses an opaque predicate from the opaque predicate library.

Overload Names - Renames methods so that as many methods as possible have the same name. This obfuscation is based on Tyma’s obfuscation.

Parameter Aliases - Adds a global field to each class in an application and assigns that field to a formal parameter in a random method of the class.

Promote Primitive Registers - Promotes all the local variables in a method to objects.

Promote Primitive Types - Changes all primitive into instances of the respective wrapper classes.

Publicizer - Makes all the fields and methods in a class public.

Random Dead Code - Adds bogus statements onto the end of a method. The appended code may include a variety of other instruction including return instructions.

Rename Registers - Renames local variables to random identifiers.

Reorder Instructions - Reorders instructions within a basic block.

Reorder Parameters - Shuffles the argument orders for all methods.

Split Classes - Splits a class such that every object becomes two objects which are linked together on a bogus field.

Static Method Bodies - Splits all of the non-static methods into a static helper method and a non-static stub that calls it.

String Encoder - Replaces strings by ‘encrypted’ versions.
**Variable Reassigner** - Rearranges the local variable table usage. Local variables that used to share a slot in the table may not any more, and new variables may share space.
APPENDIX C

SPECint-2000 Benchmark Suite Configuration File

VENDOR = Compaq
action = validate
tune = base
output_format = asc
ignore_errors = yes
ext = exe

# Compiler selection
# -Bd is NT-speak for "-v", verbose driver output

default=default=default=default:
FC = f90 -Bd
CC = cc -Bd
CXX = cc -Bd

# Baseline optimization.

default=default=default=default:
OPTIMIZE = -W3 -O0 -Ot -DWIN32 -DNDEBUG -D_CONSOLE -D_MBCS -YX -FD
FLOPTIMIZE = -fast -optimize:5 -traceback
CLOPTIMIZE = -W3 -O0 -Ot -DWIN32 -DNDEBUG -D_CONSOLE -D_MBCS -YX -FD
CXXOPTIMIZE = -W3 -O0 -Ot -DWIN32 -DNDEBUG -D_CONSOLE -D_MBCS -YX -FD

fp=base=default=default:
ONESTEP = no
# Portability flags

176.gcc=default=default=default:
EXTRA_CFLAGS = -Dalloca=_alloca -Op
EXTRA_LDFLAGS = -F10000000

178.galgel=default=default=default:
EXTRA_FFLAGS = -fixed
LDOPT = -Fe$@ -link -stack:30000000
#OBJOPT= -Fo@

186.crafty=default=default=default:
EXTRA_CFLAGS = -DNT_i386

252.eon=default=default=default:
SOURCE_PREFIX_CXX = -TP

253.perlbmk=default=default=default:
EXTRA_CFLAGS = -DSPEC_CPU2000_NTOS -DPERLDLL /MT

254.gap=default=default=default:
EXTRA_CFLAGS=-DSYS_HAS_MALLOC_PROTO -DSYS_HAS_CALLOC_PROTO

# Benchmark-specific workarounds
# Work possible compiler bugs by turning down opt level.
# In a real submission, would not be able to do this (a valid run
# uses the same baseline switches for all benchmarks).
#
254.gap=default=default=default:
COPTIMIZE=-Zi -Od
APPENDIX D

Factorial and Fibonacci Programs

```java
public class IterFact{
    public static void main(String[] args){
        System.out.println("15! = " + fact(15));
    }

    public static long fact(long x){
        int factorial = 1;
        for(int i=1, i <= x; i++){
            factorial *= i;
        }
        return factorial;
    }
}
```

```java
public class RecurFact{
    public static void main(String[] args){
        System.out.println("15! = " + fact(15));
    }

    public static long factorial(long x){
        if(x == 1)
            return x;
        return (x * factorial(x-1));
    }
}
```
```java
public class IterFib{
    public static void main(String [] args){
        System.out.println("Fib(10) = " + fib(10));
    }

    public static long fib(long n){
        int fib_1 = 0;
        int fib_2 = 1;
        int fib = 0;
        for (int i = 2; i <= n; i++){
            fib = fib_1 + fib_2;
            fib_1 = fib_2;
            fib_2 = fib;
        }
        return fib;
    }
}

public class RecurFib{
    public static void main(String [] args){
        System.out.println("Fib(10) = " + fib(10));
    }

    public static long fib(long n){
        if (n == 0 || n == 1)
            return n;
        else
            return fib(n-1) + fib(n-2);
    }
}
```
APPENDIX E

Birthmark Similarity Results

<table>
<thead>
<tr>
<th>Programs</th>
<th>CVFV</th>
<th>SMC</th>
<th>IS</th>
<th>UC</th>
<th>ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.1: Similarity results for the independence experiment using the TaNaMM birthmark techniques.

<table>
<thead>
<tr>
<th>Programs</th>
<th>CVFV</th>
<th>SMC</th>
<th>IS</th>
<th>UC</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>IterFact, IterFib</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IterFact, RecurFib</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IterFact, decode</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>IterFact, fft</td>
<td>1</td>
<td>.67</td>
<td>1</td>
<td>1</td>
<td>.92</td>
</tr>
<tr>
<td>IterFact, wc</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>RecurFact, IterFib</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>RecurFact, RecurFib</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>RecurFact, decode</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>RecurFact, fft</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>RecurFact, wc</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IterFib, decode</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>IterFib, fft</td>
<td>1</td>
<td>.67</td>
<td>1</td>
<td>1</td>
<td>.92</td>
</tr>
<tr>
<td>IterFib, wc</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>RecurFib, decode</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>RecurFib, fft</td>
<td>1</td>
<td>.67</td>
<td>1</td>
<td>1</td>
<td>.92</td>
</tr>
<tr>
<td>RecurFib, wc</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>decode, fft</td>
<td>1</td>
<td>.96</td>
<td>1</td>
<td>.98</td>
<td>.73</td>
</tr>
<tr>
<td>decode, wc</td>
<td>.2</td>
<td>.24</td>
<td>.67</td>
<td>.67</td>
<td>.45</td>
</tr>
<tr>
<td>fft, wc</td>
<td>1</td>
<td>.96</td>
<td>1</td>
<td>.98</td>
<td>.74</td>
</tr>
</tbody>
</table>

Table E.2: Similarity results for false positive experiment using the TaNaMM birthmarking techniques.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>CVFV</th>
<th>SMC</th>
<th>IS</th>
<th>UC</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.3: Similarity results for the resistance experiment using the TaNaMM birthmarking technique and the IterFact application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>CVFV</th>
<th>SMC</th>
<th>IS</th>
<th>UC</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.4: Similarity results for the resistance experiment using the TaNaMM birthmarking technique and the IterFib application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>CVFV</th>
<th>SMC</th>
<th>IS</th>
<th>UC</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.5: Similarity results for the resistance experiment using the TaNaMM birthmarking technique and the RecurFact application.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>CVFV</th>
<th>SM C</th>
<th>IS</th>
<th>UC</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.98</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.92</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>1</td>
<td>.92</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>1</td>
<td>.92</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>1</td>
<td>.96</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>.93</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>1</td>
<td>.93</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>.93</td>
<td>1</td>
<td>1</td>
<td>.98</td>
</tr>
</tbody>
</table>

Table E.6: Similarity results for the resistance experiment using the TaNaMM birthmarking technique and the RecurFib application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>CVFV</th>
<th>SM C</th>
<th>IS</th>
<th>UC</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>.8</td>
<td>.9</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>1</td>
<td>.9</td>
<td>1</td>
<td>1</td>
<td>.98</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>.8</td>
<td>.86</td>
<td>1</td>
<td>1</td>
<td>.92</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>.8</td>
<td>.9</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>.8</td>
<td>.92</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>.4</td>
<td>.9</td>
<td>1</td>
<td>1</td>
<td>.83</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>.91</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>.8</td>
<td>.9</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>.9</td>
<td>1</td>
<td>1</td>
<td>.98</td>
</tr>
</tbody>
</table>

Table E.7: Similarity results for the resistance experiment using the TaNaMM birthmarking technique and the decode application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>CVFV</th>
<th>SM C</th>
<th>IS</th>
<th>UC</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>1</td>
<td>.97</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>1</td>
<td>.73</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.93</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.98</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.92</td>
<td>.98</td>
</tr>
</tbody>
</table>

Table E.8: Similarity results for the resistance experiment using the TaNaMM birthmarking technique and the FFT application.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>CVFV</th>
<th>SMc</th>
<th>IS</th>
<th>UC</th>
<th>TaNaMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>1</td>
<td>.71</td>
<td>1</td>
<td>1</td>
<td>.98</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>1</td>
<td>.76</td>
<td>1</td>
<td>1</td>
<td>.91</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>.67</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.92</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>1</td>
<td>.66</td>
<td>1</td>
<td>1</td>
<td>.92</td>
</tr>
<tr>
<td>Codenshield</td>
<td>.83</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.96</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>.83</td>
<td>.96</td>
<td></td>
</tr>
</tbody>
</table>

Table E.9: Similarity results for the resistance experiment using the TaNaMM birth-marking technique and the wc application.

<table>
<thead>
<tr>
<th>Programs</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>10</td>
<td>38.1</td>
<td>27.8</td>
<td>18.8</td>
<td>7.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>41.4</td>
<td>37.9</td>
<td>34.6</td>
<td>33.9</td>
<td>31.8</td>
<td>30</td>
<td>27.8</td>
</tr>
</tbody>
</table>

Table E.10: Results from the independence experiment using the K-Gram birth-marking technique and $2 \leq k \leq 8$. 
<table>
<thead>
<tr>
<th>Programs</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IterFact, IterFib</td>
<td>.93</td>
<td>.81</td>
<td>.74</td>
<td>.67</td>
<td>.6</td>
<td>.56</td>
<td>.5</td>
</tr>
<tr>
<td>IterFact, RecurFib</td>
<td>.44</td>
<td>.41</td>
<td>.38</td>
<td>.37</td>
<td>.35</td>
<td>.33</td>
<td>.31</td>
</tr>
<tr>
<td>IterFact, decode</td>
<td>.52</td>
<td>.3</td>
<td>.13</td>
<td>.05</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IterFact, lt</td>
<td>.67</td>
<td>.48</td>
<td>.25</td>
<td>.14</td>
<td>.05</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IterFact, wc</td>
<td>.37</td>
<td>.15</td>
<td>.08</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RecurFact, IterFib</td>
<td>.5</td>
<td>.38</td>
<td>.28</td>
<td>.19</td>
<td>.07</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RecurFact, RecurFib</td>
<td>.45</td>
<td>.38</td>
<td>.28</td>
<td>.19</td>
<td>.07</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RecurFact, decode</td>
<td>.55</td>
<td>.29</td>
<td>.11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RecurFact, lt</td>
<td>.59</td>
<td>.38</td>
<td>.17</td>
<td>.12</td>
<td>.07</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RecurFact, wc</td>
<td>.41</td>
<td>.19</td>
<td>.11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IterFib, decode</td>
<td>.58</td>
<td>.29</td>
<td>.09</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IterFib, lt</td>
<td>.44</td>
<td>.19</td>
<td>.10</td>
<td>.01</td>
<td>.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IterFib, wc</td>
<td>.39</td>
<td>.12</td>
<td>.06</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RecurFib, decode</td>
<td>.28</td>
<td>.14</td>
<td>.04</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RecurFib, lt</td>
<td>.28</td>
<td>.21</td>
<td>.12</td>
<td>.08</td>
<td>.05</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RecurFib, wc</td>
<td>.24</td>
<td>.14</td>
<td>.08</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>decode, lt</td>
<td>.32</td>
<td>.17</td>
<td>.07</td>
<td>.03</td>
<td>.02</td>
<td>.01</td>
<td>0</td>
</tr>
<tr>
<td>decode, wc</td>
<td>.57</td>
<td>.22</td>
<td>.07</td>
<td>.03</td>
<td>.02</td>
<td>.01</td>
<td>.01</td>
</tr>
<tr>
<td>lt, wc</td>
<td>.36</td>
<td>.11</td>
<td>.03</td>
<td>.01</td>
<td>.01</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table E.11: Similarity results for false positive experiment using the K-Gram birthmark techniques.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SandMark Test 1</td>
<td>.89</td>
<td>.85</td>
<td>.71</td>
<td>.64</td>
<td>.55</td>
<td>.39</td>
<td>.19</td>
</tr>
<tr>
<td>SandMark Test 2</td>
<td>.96</td>
<td>.92</td>
<td>.88</td>
<td>.82</td>
<td>.76</td>
<td>.72</td>
<td>.69</td>
</tr>
<tr>
<td>SandMark Test 3</td>
<td>.41</td>
<td>.26</td>
<td>.17</td>
<td>.06</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SandMark Test 4</td>
<td>.96</td>
<td>.93</td>
<td>.88</td>
<td>.82</td>
<td>.75</td>
<td>.67</td>
<td>.63</td>
</tr>
<tr>
<td>SandMark Test 5</td>
<td>.78</td>
<td>.32</td>
<td>.29</td>
<td>.18</td>
<td>.10</td>
<td>.06</td>
<td>0</td>
</tr>
<tr>
<td>SandMark Test 6</td>
<td>.52</td>
<td>.41</td>
<td>.35</td>
<td>.36</td>
<td>.35</td>
<td>.33</td>
<td>.31</td>
</tr>
<tr>
<td>SmokeScreen</td>
<td>.89</td>
<td>.1</td>
<td>.03</td>
<td>.01</td>
<td>.01</td>
<td>.01</td>
<td>.01</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelex Klausmaster</td>
<td>.81</td>
<td>.67</td>
<td>.54</td>
<td>.41</td>
<td>.35</td>
<td>.28</td>
<td>.25</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.12: Similarity results for the resistance experiment using the K-Gram birthmarking technique and the IterFact application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SandMark Test 1</td>
<td>.91</td>
<td>.88</td>
<td>.78</td>
<td>.73</td>
<td>.68</td>
<td>.62</td>
<td>.54</td>
</tr>
<tr>
<td>SandMark Test 2</td>
<td>.97</td>
<td>.94</td>
<td>.91</td>
<td>.86</td>
<td>.82</td>
<td>.81</td>
<td>.79</td>
</tr>
<tr>
<td>SandMark Test 3</td>
<td>.33</td>
<td>.21</td>
<td>.13</td>
<td>.08</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SandMark Test 4</td>
<td>.97</td>
<td>.94</td>
<td>.91</td>
<td>.87</td>
<td>.82</td>
<td>.77</td>
<td>.74</td>
</tr>
<tr>
<td>SandMark Test 5</td>
<td>.73</td>
<td>.33</td>
<td>.34</td>
<td>.23</td>
<td>.11</td>
<td>.04</td>
<td>0</td>
</tr>
<tr>
<td>SandMark Test 6</td>
<td>.42</td>
<td>.32</td>
<td>.28</td>
<td>.27</td>
<td>.25</td>
<td>.23</td>
<td>.21</td>
</tr>
<tr>
<td>SmokeScreen</td>
<td>.79</td>
<td>.56</td>
<td>.34</td>
<td>.23</td>
<td>.18</td>
<td>.12</td>
<td>.04</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelex Klausmaster</td>
<td>.85</td>
<td>.74</td>
<td>.66</td>
<td>.57</td>
<td>.5</td>
<td>.42</td>
<td>.38</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.13: Similarity results for the resistance experiment using the K-Gram birthmarking technique and the IterFib application.
<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMARK Test 1</td>
<td>.86</td>
<td>.76</td>
<td>.66</td>
<td>.63</td>
<td>.5</td>
<td>.33</td>
<td>.2</td>
</tr>
<tr>
<td>SANDMARK Test 2</td>
<td>.96</td>
<td>.9</td>
<td>.84</td>
<td>.75</td>
<td>.64</td>
<td>.58</td>
<td>.5</td>
</tr>
<tr>
<td>SANDMARK Test 3</td>
<td>.90</td>
<td>.38</td>
<td>.22</td>
<td>.19</td>
<td>.14</td>
<td>.08</td>
<td>0</td>
</tr>
<tr>
<td>SANDMARK Test 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SANDMARK Test 5</td>
<td>.86</td>
<td>.71</td>
<td>.66</td>
<td>.44</td>
<td>.29</td>
<td>.17</td>
<td>0</td>
</tr>
<tr>
<td>SANDMARK Test 6</td>
<td>.77</td>
<td>.67</td>
<td>.61</td>
<td>.56</td>
<td>.6</td>
<td>.5</td>
<td>.5</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>.91</td>
<td>.76</td>
<td>.61</td>
<td>.44</td>
<td>.29</td>
<td>.17</td>
<td>0</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>.82</td>
<td>.71</td>
<td>.61</td>
<td>.5</td>
<td>.43</td>
<td>.33</td>
<td>.3</td>
</tr>
<tr>
<td>jarg</td>
<td>.96</td>
<td>.86</td>
<td>.78</td>
<td>.69</td>
<td>.57</td>
<td>.5</td>
<td>.5</td>
</tr>
</tbody>
</table>

Table E.14: Similarity results for the resistance experiment using the K-Gram birthmarking technique and the RecurFact application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMARK Test 1</td>
<td>.9</td>
<td>.86</td>
<td>.81</td>
<td>.78</td>
<td>.68</td>
<td>.56</td>
<td>.44</td>
</tr>
<tr>
<td>SANDMARK Test 2</td>
<td>.97</td>
<td>.93</td>
<td>.88</td>
<td>.83</td>
<td>.77</td>
<td>.75</td>
<td>.72</td>
</tr>
<tr>
<td>SANDMARK Test 3</td>
<td>.59</td>
<td>.34</td>
<td>.15</td>
<td>.13</td>
<td>.09</td>
<td>.05</td>
<td>0</td>
</tr>
<tr>
<td>SANDMARK Test 4</td>
<td>.97</td>
<td>.93</td>
<td>.88</td>
<td>.83</td>
<td>.77</td>
<td>.75</td>
<td>.72</td>
</tr>
<tr>
<td>SANDMARK Test 5</td>
<td>.86</td>
<td>.66</td>
<td>.42</td>
<td>.26</td>
<td>.14</td>
<td>.06</td>
<td>0</td>
</tr>
<tr>
<td>SANDMARK Test 6</td>
<td>.72</td>
<td>.50</td>
<td>.42</td>
<td>.36</td>
<td>.32</td>
<td>.3</td>
<td>.28</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>.93</td>
<td>.83</td>
<td>.73</td>
<td>.63</td>
<td>.55</td>
<td>.45</td>
<td>.33</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>.83</td>
<td>.76</td>
<td>.65</td>
<td>.51</td>
<td>.45</td>
<td>.3</td>
<td>.30</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.15: Similarity results for the resistance experiment using the K-Gram birthmarking technique and the RecurFib application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMARK Test 1</td>
<td>.96</td>
<td>.91</td>
<td>.88</td>
<td>.86</td>
<td>.82</td>
<td>.8</td>
<td>.78</td>
</tr>
<tr>
<td>SANDMARK Test 2</td>
<td>.96</td>
<td>.96</td>
<td>.95</td>
<td>.94</td>
<td>.93</td>
<td>.91</td>
<td>.9</td>
</tr>
<tr>
<td>SANDMARK Test 3</td>
<td>.71</td>
<td>.52</td>
<td>.37</td>
<td>.28</td>
<td>.23</td>
<td>.2</td>
<td>.19</td>
</tr>
<tr>
<td>SANDMARK Test 4</td>
<td>.96</td>
<td>.92</td>
<td>.88</td>
<td>.83</td>
<td>.78</td>
<td>.73</td>
<td>.68</td>
</tr>
<tr>
<td>SANDMARK Test 5</td>
<td>.98</td>
<td>.98</td>
<td>.98</td>
<td>.98</td>
<td>.98</td>
<td>.98</td>
<td>.97</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>.91</td>
<td>.86</td>
<td>.79</td>
<td>.73</td>
<td>.67</td>
<td>.61</td>
<td>.56</td>
</tr>
<tr>
<td>Codeshield</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>.94</td>
<td>.93</td>
<td>.92</td>
<td>.86</td>
<td>.83</td>
<td>.79</td>
<td>.76</td>
</tr>
<tr>
<td>jarg</td>
<td>.96</td>
<td>.94</td>
<td>.92</td>
<td>.91</td>
<td>.9</td>
<td>.89</td>
<td>.89</td>
</tr>
</tbody>
</table>

Table E.16: Similarity results for the resistance experiment using the K-Gram birthmarking technique and the decode application.
Table E.17: Similarity results for the resistance experiment using the K-Gram birthmarking technique and the fft application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>.96</td>
<td>.94</td>
<td>.93</td>
<td>.92</td>
<td>.91</td>
<td>.91</td>
<td>.9</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>.96</td>
<td>.95</td>
<td>.94</td>
<td>.94</td>
<td>.93</td>
<td>.92</td>
<td></td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>.45</td>
<td>.22</td>
<td>.11</td>
<td>.07</td>
<td>.05</td>
<td>.04</td>
<td>.03</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>.98</td>
<td>.96</td>
<td>.92</td>
<td>.89</td>
<td>.86</td>
<td>.83</td>
<td>.8</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>.93</td>
<td>.19</td>
<td>.09</td>
<td>.02</td>
<td>.01</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>.19</td>
<td>.14</td>
<td>.12</td>
<td>.11</td>
<td>.09</td>
<td>.09</td>
<td>.09</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>.99</td>
<td>.73</td>
<td>.65</td>
<td>.58</td>
<td>.48</td>
<td>.42</td>
<td>.37</td>
</tr>
<tr>
<td>Codenhl</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zeus Klassmaster</td>
<td>.94</td>
<td>.92</td>
<td>.91</td>
<td>.86</td>
<td>.87</td>
<td>.86</td>
<td>.83</td>
</tr>
<tr>
<td>jarg</td>
<td>.92</td>
<td>.88</td>
<td>.84</td>
<td>.82</td>
<td>.8</td>
<td>.79</td>
<td>.77</td>
</tr>
</tbody>
</table>

Table E.18: Similarity results for the resistance experiment using the K-Gram birthmarking technique and the wc application.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>k = 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANDMark Test 1</td>
<td>.1</td>
<td>.91</td>
<td>.93</td>
<td>.91</td>
<td>.86</td>
<td>.86</td>
<td>.81</td>
</tr>
<tr>
<td>SANDMark Test 2</td>
<td>.98</td>
<td>.95</td>
<td>.91</td>
<td>.86</td>
<td>.84</td>
<td>.81</td>
<td>.76</td>
</tr>
<tr>
<td>SANDMark Test 3</td>
<td>.65</td>
<td>.65</td>
<td>.48</td>
<td>.44</td>
<td>.38</td>
<td>.34</td>
<td>.3</td>
</tr>
<tr>
<td>SANDMark Test 4</td>
<td>.99</td>
<td>.96</td>
<td>.94</td>
<td>.9</td>
<td>.86</td>
<td>.84</td>
<td>.84</td>
</tr>
<tr>
<td>SANDMark Test 5</td>
<td>.81</td>
<td>.63</td>
<td>.53</td>
<td>.46</td>
<td>.44</td>
<td>.41</td>
<td>.34</td>
</tr>
<tr>
<td>SANDMark Test 6</td>
<td>.68</td>
<td>.56</td>
<td>.48</td>
<td>.43</td>
<td>.41</td>
<td>.38</td>
<td>.35</td>
</tr>
<tr>
<td>Smokescreen</td>
<td>.92</td>
<td>.84</td>
<td>.77</td>
<td>.72</td>
<td>.67</td>
<td>.64</td>
<td>.61</td>
</tr>
<tr>
<td>Codenhl</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zeus Klassmaster</td>
<td>.95</td>
<td>.91</td>
<td>.88</td>
<td>.84</td>
<td>.81</td>
<td>.79</td>
<td>.76</td>
</tr>
<tr>
<td>jarg</td>
<td>.91</td>
<td>.87</td>
<td>.82</td>
<td>.78</td>
<td>.75</td>
<td>.72</td>
<td>.7</td>
</tr>
</tbody>
</table>

Table E.19: Results from the independence experiment using the WPP birthmarking technique.

<table>
<thead>
<tr>
<th>Program</th>
<th>WPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>69.2%</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>63.6%</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Programs</th>
<th>WPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IterFact, IterFib</td>
<td>.08</td>
</tr>
<tr>
<td>IterFact, RecurFib</td>
<td>.08</td>
</tr>
<tr>
<td>IterFact, decode</td>
<td>.54</td>
</tr>
<tr>
<td>IterFact, fft</td>
<td>.30</td>
</tr>
<tr>
<td>IterFact, wc</td>
<td>.30</td>
</tr>
<tr>
<td>RecurFib, IterFib</td>
<td>.64</td>
</tr>
<tr>
<td>RecurFib, RecurFib</td>
<td>.09</td>
</tr>
<tr>
<td>RecurFib, decode</td>
<td>.4</td>
</tr>
<tr>
<td>RecurFib, fft</td>
<td>.45</td>
</tr>
<tr>
<td>RecurFib, wc</td>
<td>.45</td>
</tr>
<tr>
<td>IterFib, decode</td>
<td>.64</td>
</tr>
<tr>
<td>IterFib, fft</td>
<td>.36</td>
</tr>
<tr>
<td>RecurFib, decode</td>
<td>.34</td>
</tr>
<tr>
<td>RecurFib, fft</td>
<td>.13</td>
</tr>
<tr>
<td>RecurFib, wc</td>
<td>.02</td>
</tr>
<tr>
<td>decode, fft</td>
<td>.07</td>
</tr>
<tr>
<td>decode, wc</td>
<td>.15</td>
</tr>
<tr>
<td>fft, wc</td>
<td>.14</td>
</tr>
</tbody>
</table>

Table E.20: Similarity results for false positive experiment using the WPP birthmark technique.

<table>
<thead>
<tr>
<th>Obfuscation</th>
<th>IterFact</th>
<th>IterFib</th>
<th>RecurFact</th>
<th>RecurFib</th>
<th>decode</th>
<th>fft</th>
<th>wc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM Test 1</td>
<td>1</td>
<td>1</td>
<td>.45</td>
<td>.26</td>
<td>1</td>
<td>.11</td>
<td>1</td>
</tr>
<tr>
<td>SM Test 1a</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SM Test 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.95</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SM Test 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SM Test 4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.24</td>
<td>.03</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SM Test 4a</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SM Test 5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.99</td>
<td>1</td>
</tr>
<tr>
<td>SM Test 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SmokeScreen</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Codenaked</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Zelix Klassmaster</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.02</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>jarg</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.08</td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.21: Similarity results for the resistance experiment using the WPP birthmarking technique.
APPENDIX F

Complete Publications List


REFERENCES


Conference on the Centre for Advanced Studies on Collaborative Research, 1999.
