The Use of Software-Based Integrity Checks in Software Tamper Resistance Techniques

Ginger Myles

gmyles@us.ibm.com

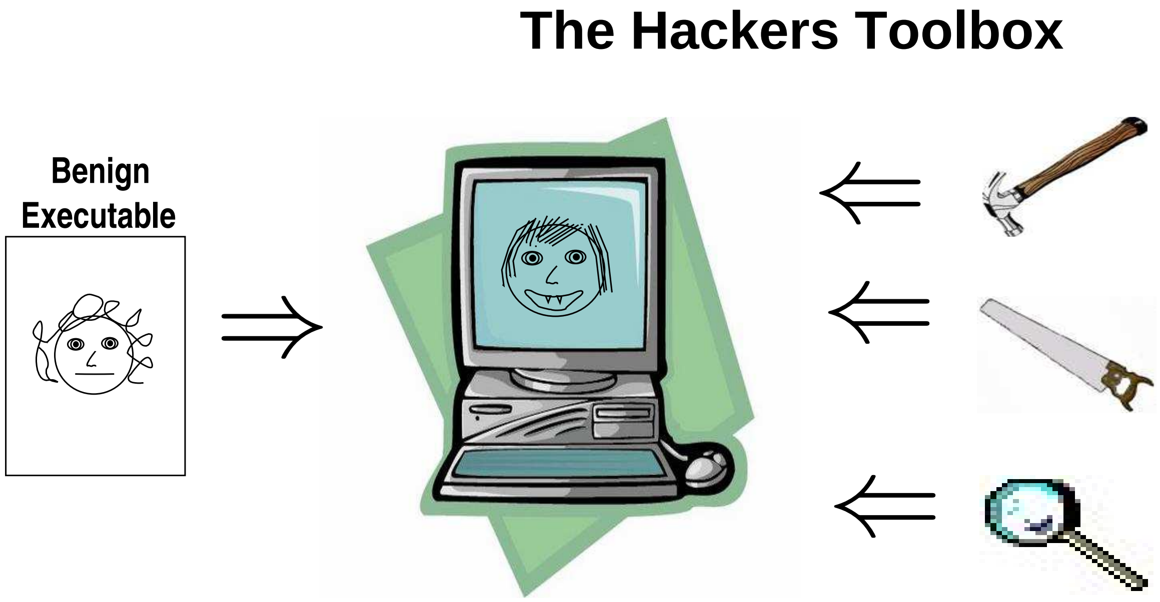
IBM Almaden Research Center

Talk Overview

- Integrity check overview
- Use of integrity checks in tamper resistance techniques
- Attack on integrity checks

Attack Model

- Software executing on a potentially hostile host.
- Adversary has full control over the software's execution.
- Adversary can use any program analysis tools to detect and circumvent the checks.

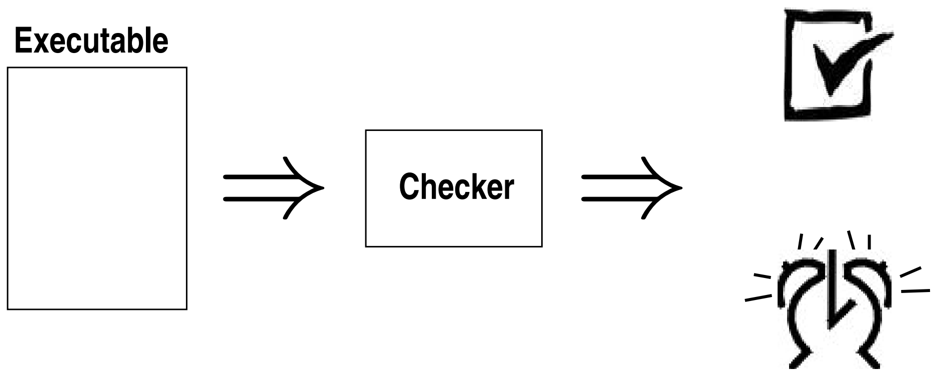


Software Tamper Resistance

- Detect that the program has been altered.
- Cause the program to fail when tampering has been detected.

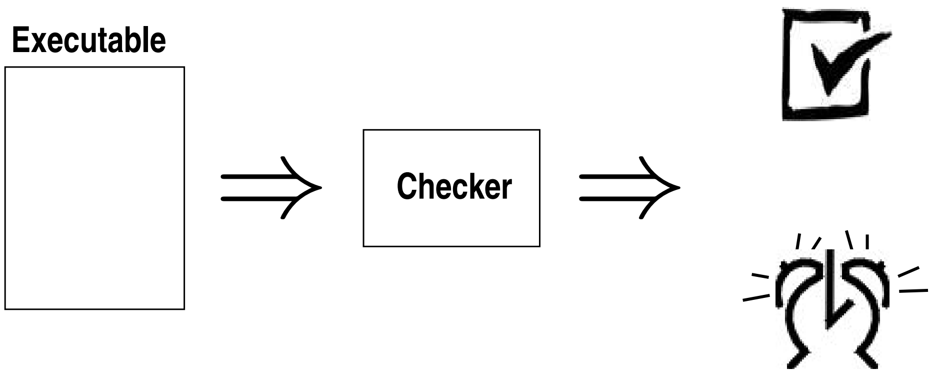
Integrity Check Overview

- What are they?
  - A mechanism used to identify the integrity of the program and/or the environment in which it is executing.



Integrity Check Overview

- What are they?
  - Static - integrity is checked only once during start-up
  - Dynamic - integrity is checked repeatedly as the program executes



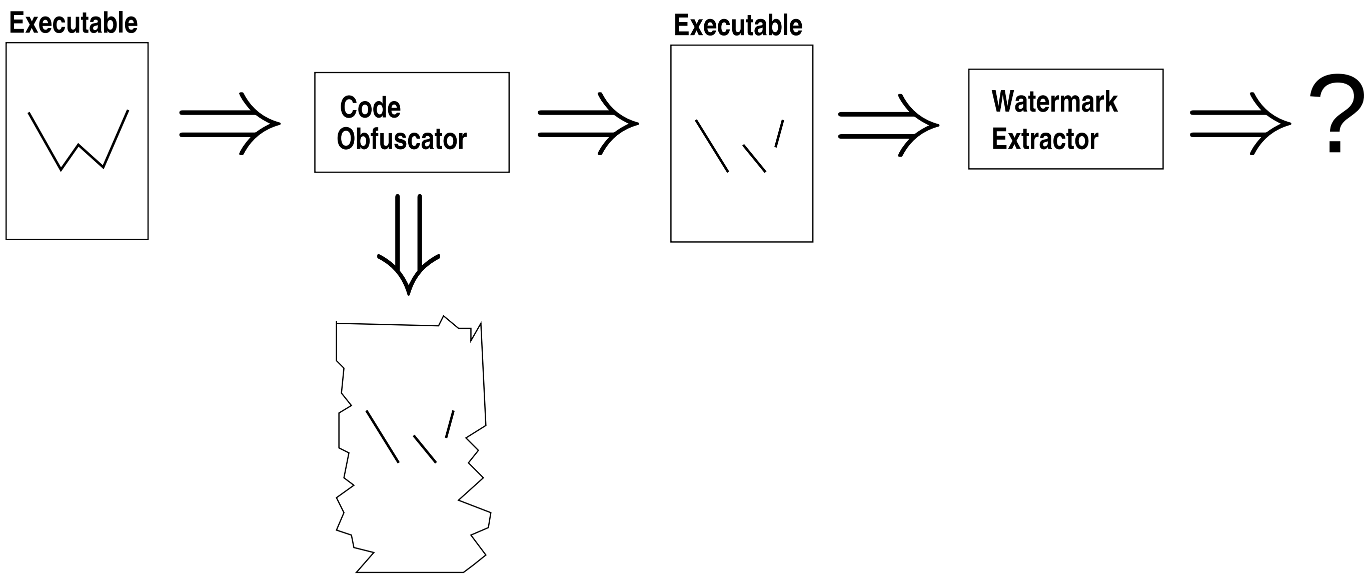
Integrity Check Overview

- How are they used?
  - Generally used as part of a larger tamper resistance scheme.

Assertion Check Based (BAD!)	Use Based (Better)
`ic = performCheck();`	`ic = performCheck();`
`if(ic != PREDICTED_VALUE) {fail;}`	`decryptSegment(ic, k);`

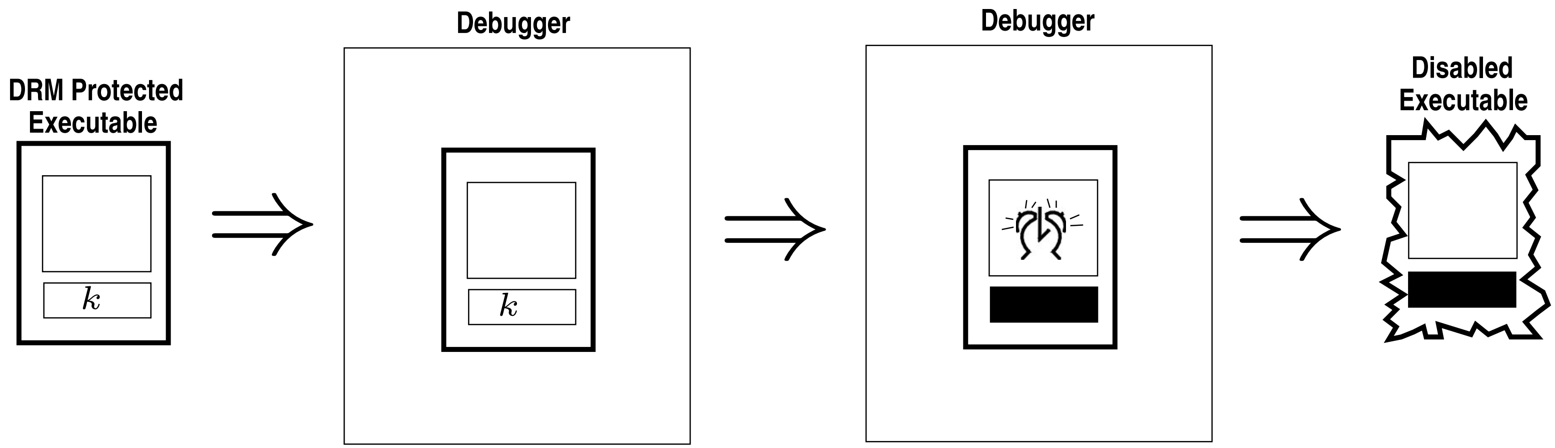
Integrity Check Overview

- Why are they used?
  - Program integrity verification
    - Prevent license check removal
    - Protect a watermark from damage or removal



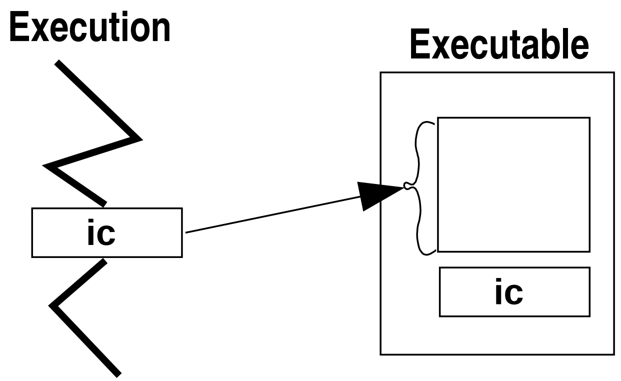
Integrity Check Overview

- Why are they used?
  - Environmental integrity verification
    - Detect the use of debuggers or emulators
      - Protect the keys embedded in DRM systems
      - Sensitive code that it is only decrypted for execution



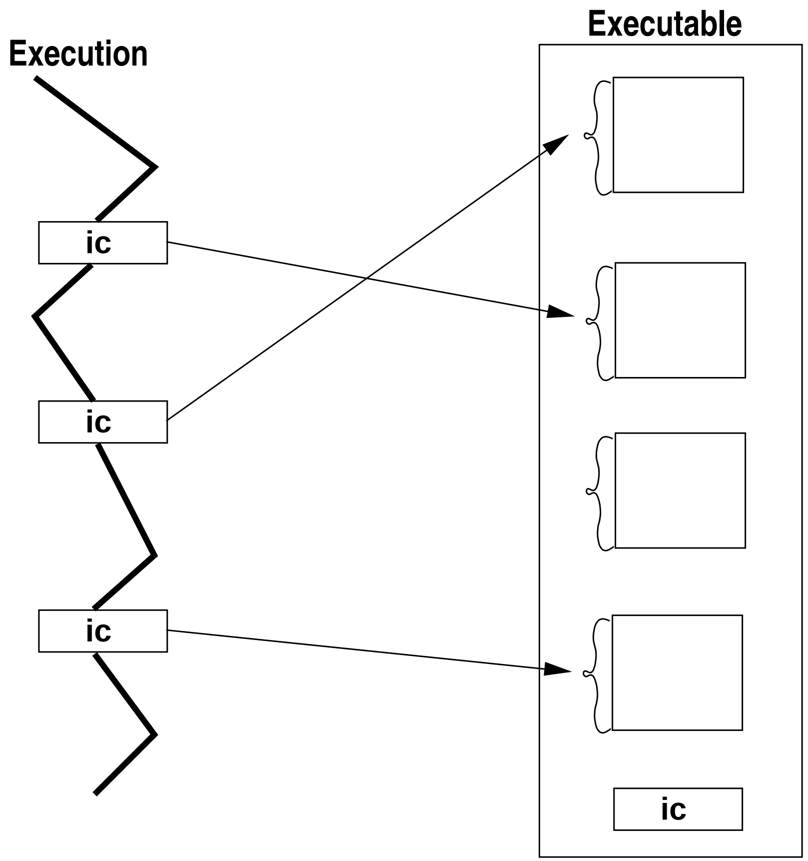
Program Integrity Verification

- Code Checksum
  - Probably the oldest method
  - Straight forward implementation
  - Quite efficient



Program Integrity Verification

- Code Checksum – The Negatives
  - Reading the code segment is generally atypical.

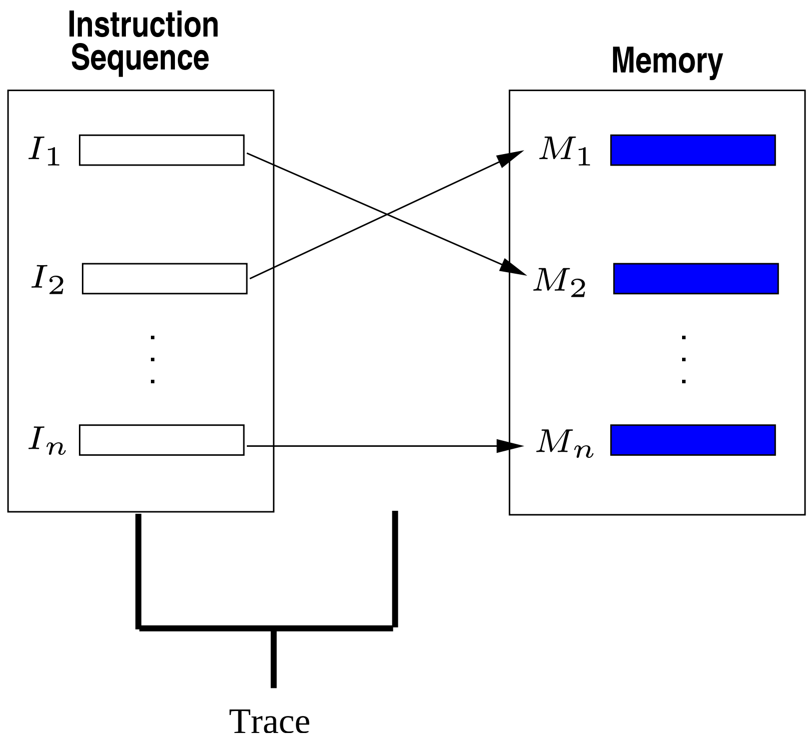


Program Integrity Verification

- Code Checksum – The Negatives
  - Hackers pinpoint checks through breakpoints or code inspection.
  - Only verifies static properties.
    - May not detect temporary instruction patches or other run-time attacks.

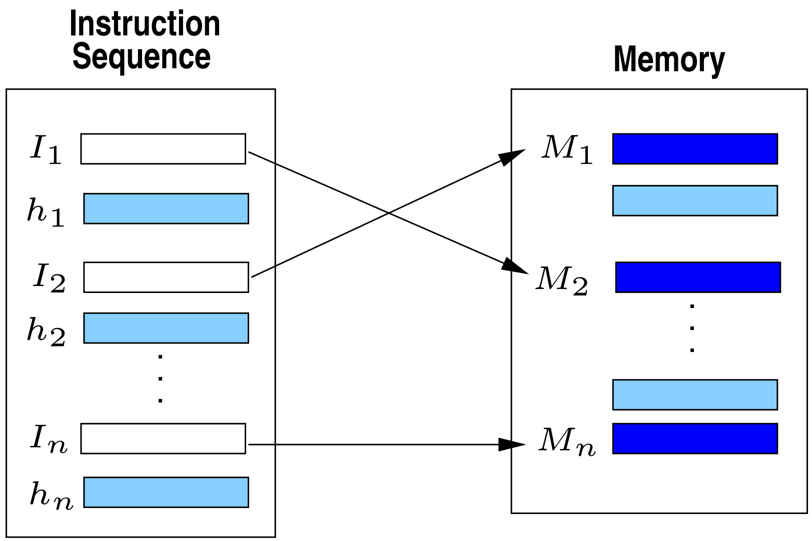
Program Integrity Verification

- Oblivious Hashing [Chen et al., 2002]
  - Dynamic - based on execution trace.
  - Monitor both instructions and memory references.
  - Compute hash value from execution trace.



Program Integrity Verification

- Oblivious Hashing
  - Hashing locations must be extensive and spread throughout program.
  - To protect a function all functions on the calling-hierarchy must be protected.
  - Trace should include memory references made by each instruction and the instruction itself.



Environment Integrity Verification

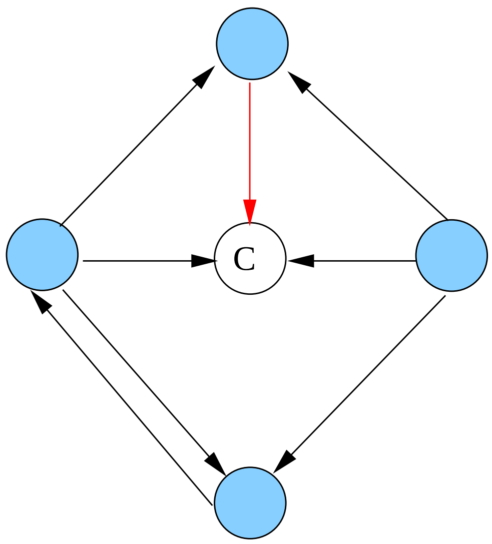
- Detection of debuggers and other similar simulation tools
  - Use a checksum to detect a breakpoint.
  - Measure elapsed time to execute a sequence of instructions.
  - Look for tool specific hooks.

Environment Integrity Verification

- The Negatives
  - Level of checksum granularity will effect success of detecting a breakpoint.
  - Detection is tool specific.

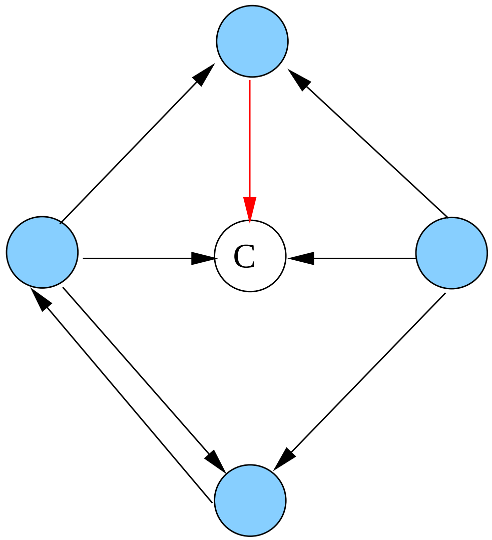
Real Uses of Integrity Checks

- Check and Guard System [Chang and Atallah, 2001]
  - Network of guards.
  - Each guard responsible for performing some type of integrity check.
  - Protect each other and the program in an interlocking fashion.
  - Some guards can repair altered code.



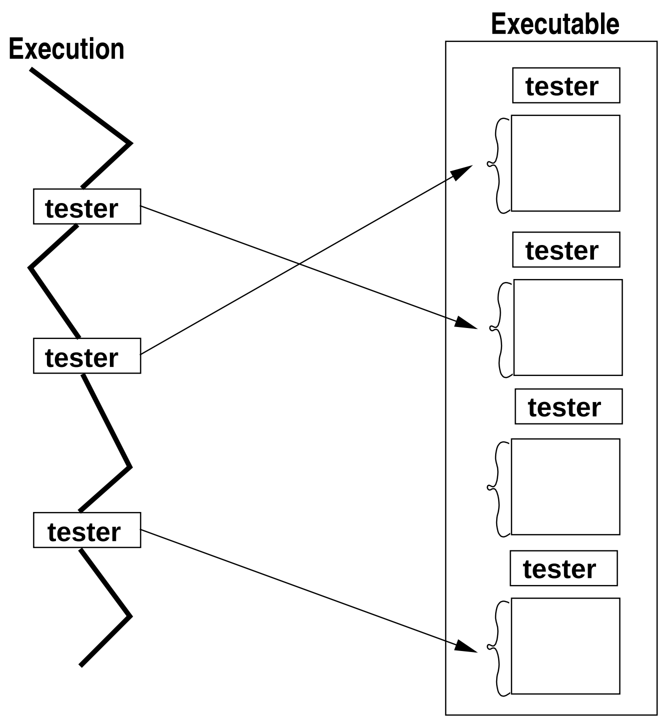
Real Uses of Integrity Checks

- Check and Guard System
  - Repairing guard inserted prior to code.
  - Checksumming guard inserted at a point when code will be present in program image.
  - Strongly connected guard graph increases the efforts required by the attacker.



Real Uses of Integrity Checks

- Testers and Correctors [Horne et al., 2002]
  - Collection of testers which each hash a single contiguous section of code.
  - Testers are sprinkled throughout code and triggered by normal program execution.

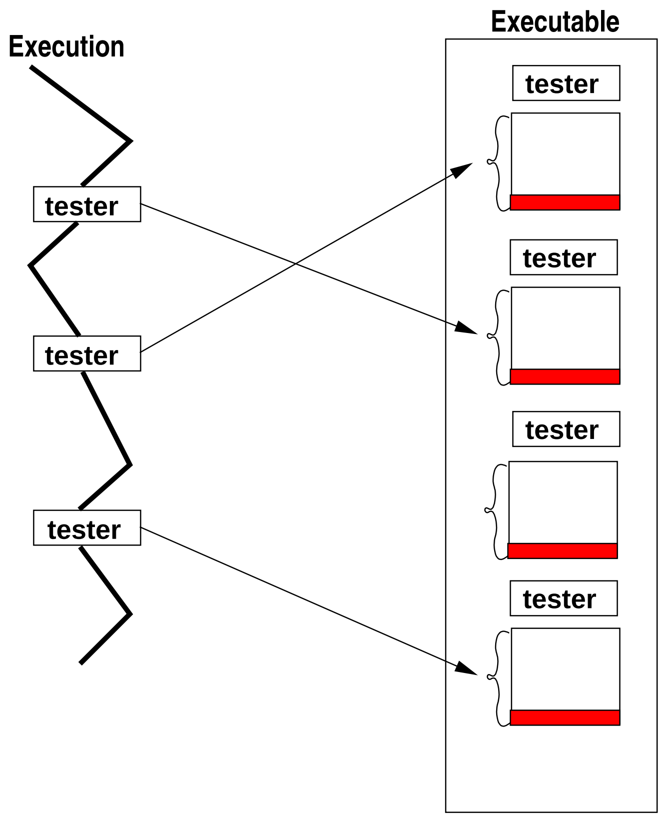


Real Uses of Integrity Checks

- Testers and Correctors [Horne et al., 2002]
  - Compares hash value with correct value.
  - Incorrect value triggers response mechanism via simple function call.

Real Uses of Integrity Checks

- Testers and Correctors
  - Each hash interval contains a corrector.
  - Corrector set to a value such that the interval hashes to a fixed value.

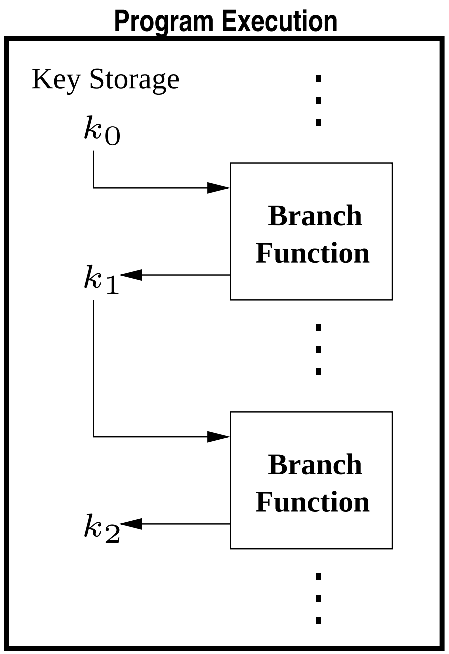


Real Uses of Integrity Checks

- Branch-Based Tamper Resistance [Myles, Jin, 2005]
  - Based on Branch Function obfuscation [Linn and Debray, 2003]:
    - Designed to disrupt static disassembly.
    - Exploits assumption that a function call returns to the instruction immediately following the call instruction.
    - Execution is rerouted through the branch function.
    - The correct target is identified based on the call location.
      - $T[h(j_i)] = t_i - j_i$
    - Return address on the stack is overwritten.

Real Uses of Integrity Checks

- Branch-Based Tamper Resistance



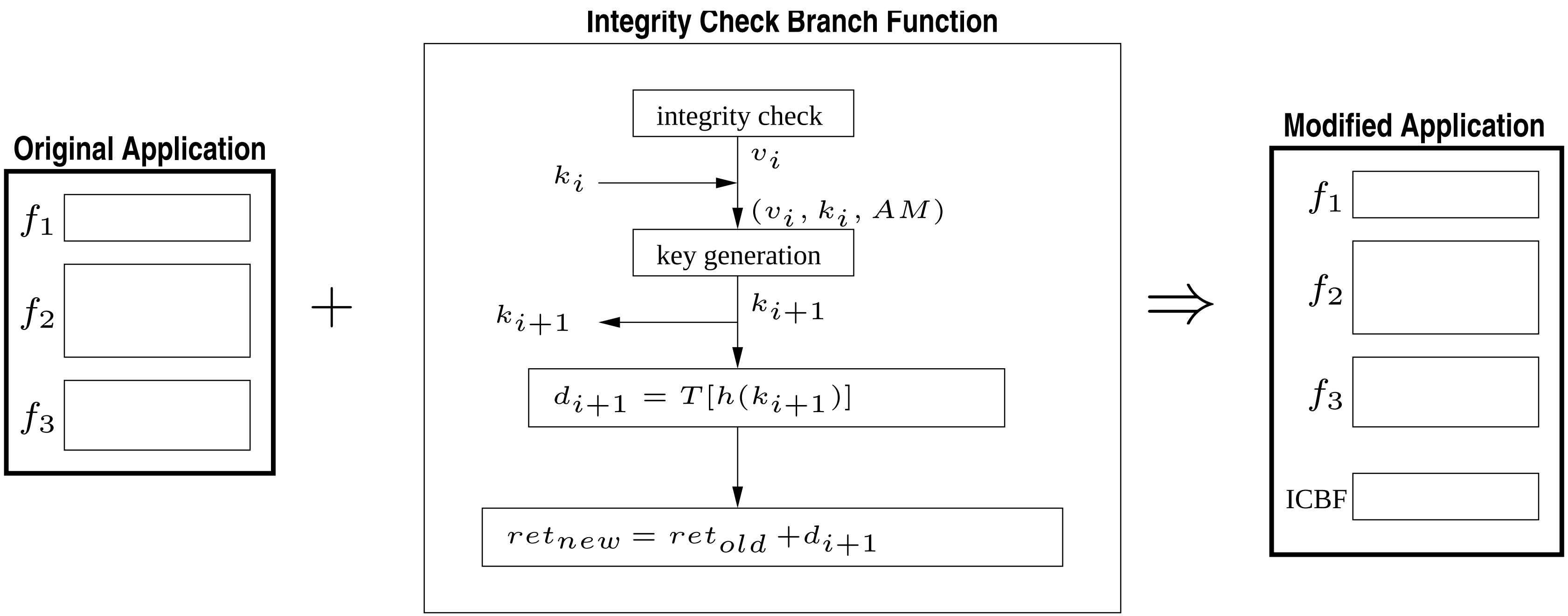
- Key: Link proper program execution and the key evolution.

Real Uses of Integrity Checks

- Branch-Based Tamper Resistance
  - Integrity Check Branch Functions
    1. Perform an integrity check of the program or environment producing the value v_i.
    2. Generate the next key using a secure one-way hash function, the previous key, and the integrity check value.
       - $k_{i+1} = SHA1(v_i, k_i)$
    3. Use k_{i+1} to identify the instruction where execution will resume.

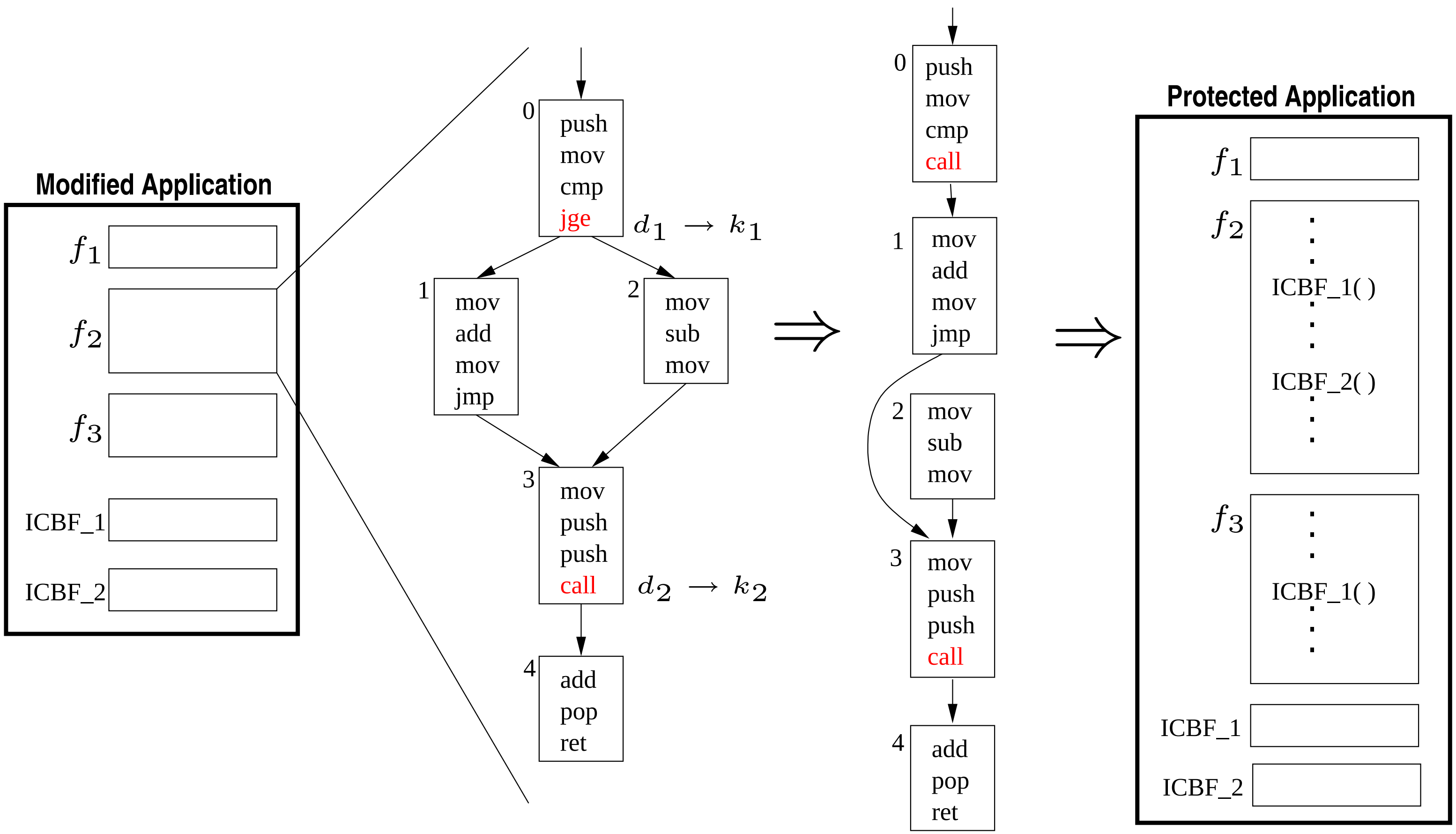
Real Uses of Integrity Checks

- Branch-Based Tamper Resistance
  - Integrity Check Branch Function Construction



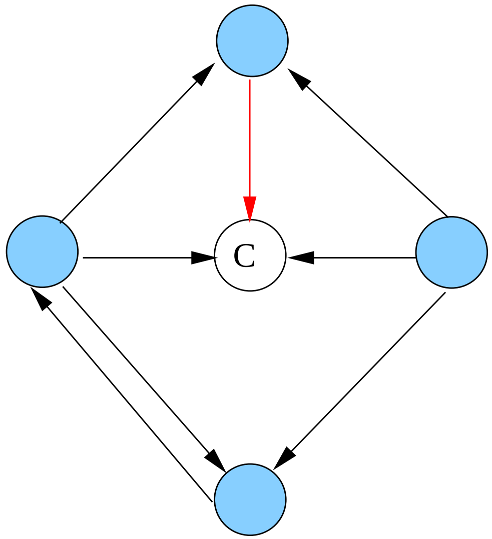
Real Uses of Integrity Checks

- Branch-Based Tamper Resistance
  - Branch Instruction Replacement



Real Uses of Integrity Checks

- Branch-Based Tamper Resistance
  - Able to construct an intertwined network of ICBF’s



Attacks on Integrity Checks

- Circumvention of self-hashing has been accomplished on UltraSparc, x86, PowerPC, AMD64, and ARM architectures [van Oorschot et al., 2005].
  - Implicit assumption that a data read from memory address x is the same as an instruction fetch from x.
  - $I(x) \neq D(x)$ will verify code that is never executed and the executed is never checked.
  - Manipulate virtual to physical address mappings such that each virtual address refers to two different physical addresses
    - code references and data references
  - Done through segmentation and translation lookaside buffers.

Wrap-up

- We looked at a common type of integrity checking
  - how it is used in real tamper resistance techniques and
  - how it can be circumvented.

Wrap-up

- Questions I'm considering:
  - Is there hope for strictly software-based techniques?
  - Is there a way we can determine the level of protection provided by the different types of integrity checks?

Wrap-up

- Questions I’m considering:
  - Is there hope for strictly software-based techniques?
  - Is there a way we can determine the level of protection provided by the different types of integrity checks?
    - Begin building an incremental strength evaluation scheme for software tamper resistance techniques.

Wrap-up

- Questions I'm considering:
  - Is there hope for strictly software-based techniques?
  - Is there a way we can determine the level of protection provided by the different types of integrity checks?
    - Begin building an incremental strength evaluation scheme for software tamper resistance techniques.
    - With the Check and Guard system or the Branch-Based technique strength can be customized or checks can be replaced once discovered they are breakable.

References

- Protecting Software Code by Guards, *Chang and Atallah*, Proc. of 1st ACM Workshop on Security and Privacy in Digital Rights Management, 2001.
- Oblivious Hashing: A Stealthy Software Integrity Verification Primitive, *Chen, Venkatesan, Cary, Pang, Sinha, and Jakubowski*, Proc. of 5th International Workshop on Information Hiding, 2002.
- Towards Better Software Tamper Resistance, *Jin and Myles*, Proc. of Information Security: 8th International Conference, 2005.
- Self-Validating Branch-Based Software Watermarking, *Myles and Jin*, Proc. of 7th International Workshop on Information Hiding, 2005.
- Hardware-Assisted Circumvention of Self-Hashing Software Tamper Resistance, *van Oorschot, Somayaji, and Wurster*, IEEE Transactions on Dependeble and Secure Computing, 2005.