Software-Based Attestation

Software-Root of Trust

History, Constructions, Applications

Adrian Perrig

CyLab / Carnegie Mellon University
Is my computer secure?
Dynamic Root of Trust

- CPU instruction (SKINIT/SENTER) accepts a memory region as input and atomically:
  - Resets dynamic PCRs
  - Disables interrupts
  - Protects memory region
  - Extends a measurement of the region into PCR 17
  - Begins executing at the start of the memory region

Goal of Software-based Attestation:
Achieve a Dynamic Root of Trust without HW support

- Properties
  - Isolation of memory region
  - Remote attestation of code integrity and start of execution
Software-based Attestation

• Desired property: externally verifiable code execution without special HW

• Why not simply rely on HW?
  – Legacy devices
  – Perform attestation if vPro/SVM are flawed
  – Software-based attestation requires no secrets
  – Enables new applications
Potential Approaches & Challenges

• Only allow SW stored in ROM to execute
  – Need to exchange ROM to update SW

• ROM bootloader loads SW to execute, reboot each time
  – Impractical
  – Reboot cannot be remotely verified

• “Hashing engine” sends code hash to verifier
  – How to verify that hash was correctly computed?

• Crypto: secure multi-party computation
  – Computationally expensive to execute
Past Examples of User-Verifiable Code Execution?

• What past computing system permitted users to verify code execution?
  – Abacus
  – Balance
  – Slide rule
  – Punchcard mainframe
Initial Setting

- Untrusted device D, trusted verifier V
- V knows expected memory contents of D
- V wants to obtain proof of D’s memory contents, obtain memory integrity
- D executes verification function VF
- Challenge: VF may be malicious and return expected result!
_strawman verification function (1)_

- **Approach 1:** Verifier asks device to compute a cryptographic hash function over memory
  - \( V \rightarrow D: \text{Checksum request} \)
  - \( D \rightarrow V: \text{SHA-1(Memory)} \)

- **Attack:** malicious code pre-computes and replays correct hash value

[Diagram showing Code, Checksum Code (0..0), and Unused memory]
Strawman Verification Function (2)

- Approach 2: Verifier picks a random challenge, device computes Message Authentication Code (MAC) using challenge as a key
  - V → D: Checksum request, random K
  - D → V: HMAC-SHA-1( K, Memory )
- **Attack**: Malicious code computes correct checksum over expected memory content
Presentation Outline

- History of software-based attestation mechanisms
- SWATT verification function
- Pioneer: externally-verifiable code execution
- Applications of software-based attestation
- Open research challenges, conclusion
Reflection

• Reflection as a Mechanism for Software Integrity Verification

• Reflection: SW that verifies its own operation
  – Relies on difficulty of predicting and monitoring modern processor behavior, and fills entire memory with high-entropy data

• Verification function:
  – Fill memory with random content
  – Clear system state, disable interrupts
  – Compute hash over entire memory
  – Return hash and system state

• Verifier checks duration of computation, hash, and system state
Genuinity

- Establishing the Genuinity of Remote Computer Systems
  - Kennell and Jamieson, Usenix Security Symposium, August 2003

- Verifier sends checksum code to remote system

- Verification function:
  - Randomized memory access to cause unpredictable cache misses
  - Integration of hardware parameters into checksum

- Security argument
  - Simulation of verification function much slower due to complexity of simulating architectural features (e.g., cache misses)
Soft Tamper-Proofing …

• Soft Tamper-Proofing via Program Integrity Verification in Wireless Sensor Networks
  – Park and Shin, IEEE TMC, 4(3), May/June 2005

• Verification function:
  – Reboots sensor
  – Fills data memory with random, uncompresible data
  – Executes hash function over entire memory

• Verifier device times execution and verifies hash value
Remote SW Attestation for Sensors

- Remote Software-based Attestation for Wireless Sensors
  - Shaneck, Mahadevan, Kher, and Kim; ESAS, July 2005

- Verification function:
  - Encrypted code
  - Self-modifying code
  - Randomized traversal

- Verifier uses timing to check result
Alien vs. Quine

- Alien vs. Quine, the Vanishing Circuit and Other Tales from the Industry’s Crypt
  - Gratzer and Naccache, invited paper to Eurocrypt 2006
- Assumption that code immediately starts execution after reboot
- Idea: implement a secure loader program
- Verification function:
  - Start: ldx In ; X ← In
  - bne store ; if X!=0 goto store
  - Print: lda M, X ; A ← M[X]
  - sta Out ; Out ← A
  - incx ; X++
  - bne Print ; if X !=0 goto print
  - bra Start ; goto Start
- Store: lda In ; A ← In
  - sta M, X ; M[X] ← In
  - bra Start ; goto Start
- Interesting approach for systems where verifier is in direct control
SWATT

• SWATT: SoftWare-based ATTestation for Embedded Devices
  – Seshadri, Perrig, van Doorn, Khosla; IEEE Symposium on Security and Privacy, May 2004

• Verifier sends nonce to device

• Verification function:
  – Pseudorandom memory traversal to compute memory checksum

• Verifier times checksum computation and verifies checksum

• Observations:
  – Malicious code must verify each memory access to replace memory reads of changed memory locations with expected content, resulting in a detectable time overhead
SWATT Verification Function

- Observation: time is externally detectable property that reveals tampering of checksum computation

- Approach
  - Use time as externally detectable property, compute checksum that slows down if tampering occurs
  - Compute checksum in pseudo-random order
  - Attacker needs to verify each memory access → slowdown
Assumptions

- Verifier knows hardware configuration of device
  - In particular, verifier knows processor clock speed
- Response originates from untrusted device
  - In particular, no proxy attack (D did not contact faster host)
- Attacker model: change SW, but not HW
- Optimal implementation: code cannot be optimized
  - Denali project @ HP labs provides proof of optimal implementation of short pieces of code
  - GNU superopt
  - Open challenge to prove optimality of SWATT checksum
- No algebraic optimizations
  - Checksum has to be computed in entirety
  - Given a memory change, checksum cannot be “adjusted” without recomputation
Implementation Platform

- Bosch sensor node
  - TI MSP430 microcontroller
Seed from verifier

PRG (RC4)

Address Generation

Memory Read and Transform

Compute Checksum

Generate $i^{th}$ member of random sequence using RC4

\[
zh = 2 \\
\text{ldi } zh, 0x02
\]

\[
r15 = *(x++) \\
\text{ld } r15, x+
\]

\[
yl = yl + r15 \\
\text{add } yl, r15
\]

\[
zl = \*y \\
\text{ld } zl, y
\]

\[
\*y = r15 \\
\text{st } y, r15
\]

\[
\*x = r16 \\
\text{st } x, r16
\]

\[
zl = zl + r15 \\
\text{add } zl, r15
\]

\[
zh = \*z \\
\text{ld } zh, z
\]

Generate 16-bit memory address

\[
zl = r6 \\
\text{mov } zl, r6
\]

Load byte from memory and compute transformation

\[
r0 = \*z \\
\text{lpm } r0, z
\]

\[
r0 = r0 \text{ xor } r13 \\
\text{xor } r0, r13
\]

\[
r0 = r0 + r4 \\
\text{add } r0, r4
\]

Incorporate output of hash into checksum

\[
r7 = r7 + r0 \\
\text{add } r7, r0
\]

\[
r7 = r7 << 1 \\
\text{lsl } r7
\]

\[
r7 = r7 + \text{carry_bit} \\
\text{adc } r7, r5
\]

\[
r4 = zh \\
\text{mov } r4, zh
\]
SWATT Advantage

• SWATT time advantage = running time of fastest attack code – running time of SWATT checksum code

• Verification procedure loop has 16 assembly instructions and takes 23 cycles

• Checks require “if” statements
  – Translates to compare + branch in assembly, requires 3 cycles

• Insertion of single “if” statement increases loop execution time
  – 13% increase per iteration in our implementation
SWATT Extension

• Drawback: checksum computed over entire device memory
  – Does not scale to large memory sizes
  – Memory may contain secrets
  – Memory may contain dynamic data

• Solution: design checksum function that can check small memory areas
  – Memory area being checked includes checksum function

• Challenge: introduces many new attacks!
Attack on Partial Memory Verification

- Checksum computed over small part of memory
- **Memory copy attack**: attacker computes checksum over correct copy of memory
Improved Checksum Approach

• Add chksum function execution state to checksum
  – Include program counter (PC) and data pointer
• In memory copy attack, one or both will differ from original value
• Attempts to forge PC and/or data pointer increases attacker’s execution time
ICE Assembly Code

Generate random number using T-Function
mov r15, &0x130
mov r15, &0x138
bis #0x5, &0x13A
add &0x13A, r15

Load byte from memory
add r0, r6
xor @r13+, r6

Incorporate byte into checksum
add r14, r6
xor r5, r6
add r15, r6
xor r13, r6
add r4, r6
rla r4
adc r4
Pioneer

- Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms
  - Seshadri, Luk, Shi, Perrig, van Doorn, Khosla; ACM Symposium on Operating Systems Principles (SOSP), October 2005

- First step to address untampered code execution on untrusted legacy hosts

- Implemented on Intel Pentium IV
  - Numerous challenges exist on this platform!
Challenges on x86 Platforms

• Execution time non-determinism
  – Out-of-order execution
  – Cache and virtual memory
  – Thermal effects

• Complex instruction set and architecture: how can we ensure that code is optimal?

• DMA-based attacks from malicious peripherals

• Interrupt-based attacks
  – SMM, NMI, etc.

• Attacks using exceptions

• Virtualization-based attacks
Pioneer Implementation

- Intel Xeon @ 2.8 GHz, Linux kernel 2.6.7
  - Intel Netburst Microarchitecture (Pentium 4)
  - Key: issue max 3 μops per cycle (3 way superscalar)
  - 64-bit extensions (no segmentation)
Verifiable Code Execution

- **Goal:** provide verifier with guarantee about what code executed on device

- **Approach**
  1. Verify code integrity through software-based attestation
  2. Set up untampered code execution environment
  3. Execute code
Design of Verification Function

- **Checksum Code**
- **Hash Function**
- **Target Code**

**Verification Function**
- **Measure Integrity**
- **Invoke**

**Root of Trust**
- Compute checksum
- Set up untampered execution environment
The Pioneer Protocol

- Successful verification if: $t_2 - t_1 < \text{expected time} \land \text{cksum} == \text{exp. cksum}$

- Diagram showing the protocol:
  - $t_1$: nonce, input
  - $t_2$: cksum
  - hash
  - output

- Blocks representing:
  - Checksum Code
  - Hash Function
  - Target Code
  - Output
Desired Security Property

• Verifier’s check is successful if and only if
  – Verification function is unmodified
  – Untampered execution environment is established

• Intuition: Checksum is incorrect or checksum computation slows down if attacker
  – Modifies verification function and forges correct checksum, or
  – Fakes creation of untampered code execution environment
Potential Attacks

• Execution tampering attacks
  – Running malicious OS/VMM at higher privilege level
  – Getting control through interrupts and exceptions

• Checksum forgery attacks
  – Memory-copy
  – Data substitution
  – Code optimization
  – Parallel execution
  – Exploiting superscalar architecture
  – Pre-computation/replay attacks
Exceptions and Interrupts

- Attacker installs malicious exception and interrupt handlers
- Attacker generates exception or interrupt during execution of Verification Function
- Solution: Replace interrupt and exception handlers
- At what stage of execution to replace the handlers?
Replacing Handlers

- When is it safe to replace handlers?
  - Initialization code: Attacker can skip!
  - Checksum code: Attacker can skip and save processor cycles!
  - After checksum computation: Attacker can prevent execution through debug breakpoint!
Replacing Handlers: Stack Trick
Replacing Handlers: Stack Trick

Stack Pointer

Checksum
Checksum
Checksum
Checksum

CPU

Interrupt or Exception
Results – Runtime Difference

Execution Time [ms]

Time of Measurement [minutes]

- Legitimate Code’s Expected Runtime
- Legitimate Code’s Observed Runtime
- Legitimate Code’s Runtime and Network RTT
- Correct Computation Time Threshold
- Theoretically Best Adversary’s Runtime
- Adversary’s Runtime and Network RTT

\[ .3 \text{ms} \]
Pioneer Discussion

• Verifier can obtain untampered execution guarantee for code executing on untrusted platform

• Similar attestation property to AMD SVM or Intel TXT

• Drawbacks
  – Requires defense against proxy and overclocking attack

• PioneerNG addresses many issues with multi-core CPUs, SMM, portability issues, etc.
Applications

- Rootkit detector
- Recovery from system compromise
- Key establishment
Sensor Network Key Establishment

• How to establish a shared secret?
  – Attacker may know entire memory contents of a newly shipped node
  – After a node has been compromised, attacker may have altered authentic public keys or knows secret keys
  – Without authentication Diffie-Hellman protocol is vulnerable to man-in-the-middle attack:
    • A $\rightarrow$ B: $g^a \mod p$
    • B $\rightarrow$ A: $g^b \mod p$
Problem Formulation

• Given nodes in a sensor network, how can any pair of nodes establish a shared secret without any prior authentic or secret information?

• In theory, this is impossible … because of active MitM attack

• Assumptions
  – Attacker cannot compute faster than sensor node
  – Each node has a unique, public, unchangeable identity stored at a fixed memory address
  – Secure source of random numbers
ICE Key Establishment

• Intuition: leverage ICE to compute checksum faster than any other node, and use checksum as a short-lived shared secret!

• Challenge: how to use short-lived shared secret to bootstrap long-lived secret?
  – Authenticate Diffie-Hellman public key
First Attempt

Pick random $a$
Compute $g^a \mod p$

$t_0: g^a \mod p$

$g^a \mod p = \text{challenge}$
Compute $cksum$

Pick random $b$
Compute $g^b \mod p$

$t_1: g^b \mod p, \text{MAC}(c, g^b \mod p)$
Second Attempt

Pick random $a$

Compute $g^a \mod p$

$t_0: g^a \mod p$

$g^a \mod p = \text{challenge}$

Compute $cksum \ c$

Pick random $b$

Compute $g^b \mod p$

$t_1: g^b \mod p, \text{MAC}(c, g^b \mod p)$
Goal: A and B can authenticate each other’s messages

- Pick random $v_2$
  
  - $v_1 = H(v_2)$, $v_0 = H(v_1)$
  
- one-way chain: $v_0 \leftarrow v_1 \leftarrow v_2$

Assume: A knows authentic $w_0$

- B knows authentic $v_0$

- Pick random $w_2$
  
  - $w_1 = H(w_2)$, $w_0 = H(w_1)$
  
- $w_0 \leftarrow w_1 \leftarrow w_2$
ICE Key Establishment

Pick random $a$, $g_a = g^a \mod p$
Compute $g'_a = H(g_a)$, $g''_a = H(g'_a)$, $g''_a \leftarrow g'_a \leftarrow g_a$

$t_0: g''_a \rightarrow g_a = \text{challenge}$
Compute checksum $c$

$w_0 \leftarrow w_1 \leftarrow w_2$

$t_1: w_0 \xrightarrow{\text{MAC}(c, w_0)}$
\text{random } b, \ g^b \mod p$

$w_1, g^b \mod p, \text{MAC}(w_2, g^b \mod p)$

$g_a \leftarrow g'_a \leftarrow g_a$
Summary: ICE Key Re-Establishment

• Protocol can prevent man-in-the-middle attacks without authentic information or shared secret

• Attacker can know entire memory content of both parties before protocol runs

• Forces attacker to introduce more powerful node into network, prevents remote attacks
Conclusion & Open Research Challenges

- **Software-based attestation offers exciting properties**
  - Attestation on legacy systems
  - Attestation without secrets!
  - Can support HW attestation: securely reading out TPM public key

- **Open research challenges**
  - Architecture-independent verification function
  - High time difference between attack and legitimate function
  - Formally provable properties