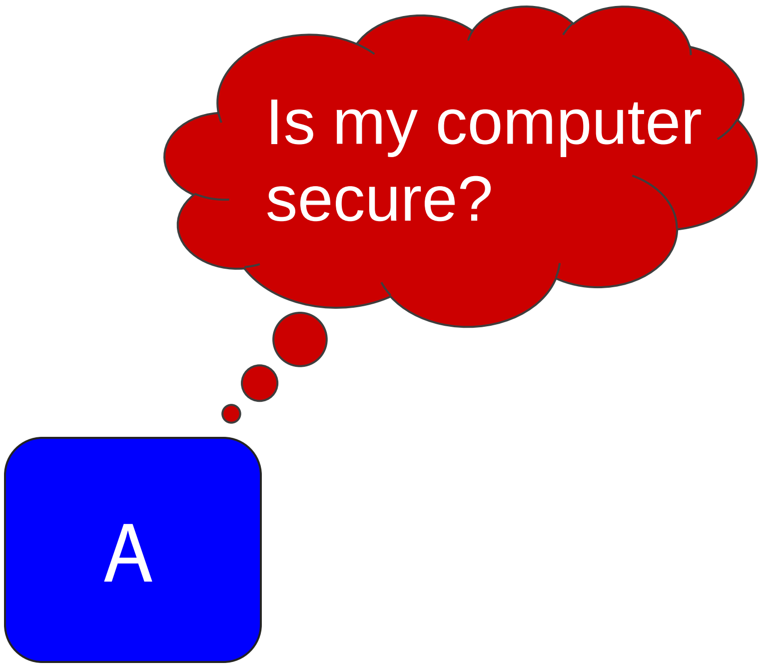
Carnegie Mellon CyLab
4616 HENRY STREET
PITTSBURGH, PA 15213
PH: (412) 268-7195
FX: (412) 268-7196
www.cylab.cmu.edu

Building Secure Applications with Attestation

Adrian Perrig
CyLab @ Carnegie Mellon University

Research in collaboration with Yanlin Li, Mark Luk, Jon McCune, Bryan Parno, Arvind Seshadri, Elaine Shi, Amit Vasudevan, Stephen Zhou
Anupam Datta, Virgil Gligor, Pradeep Khosla, Leendert van Doorn


Is my computer secure?
A

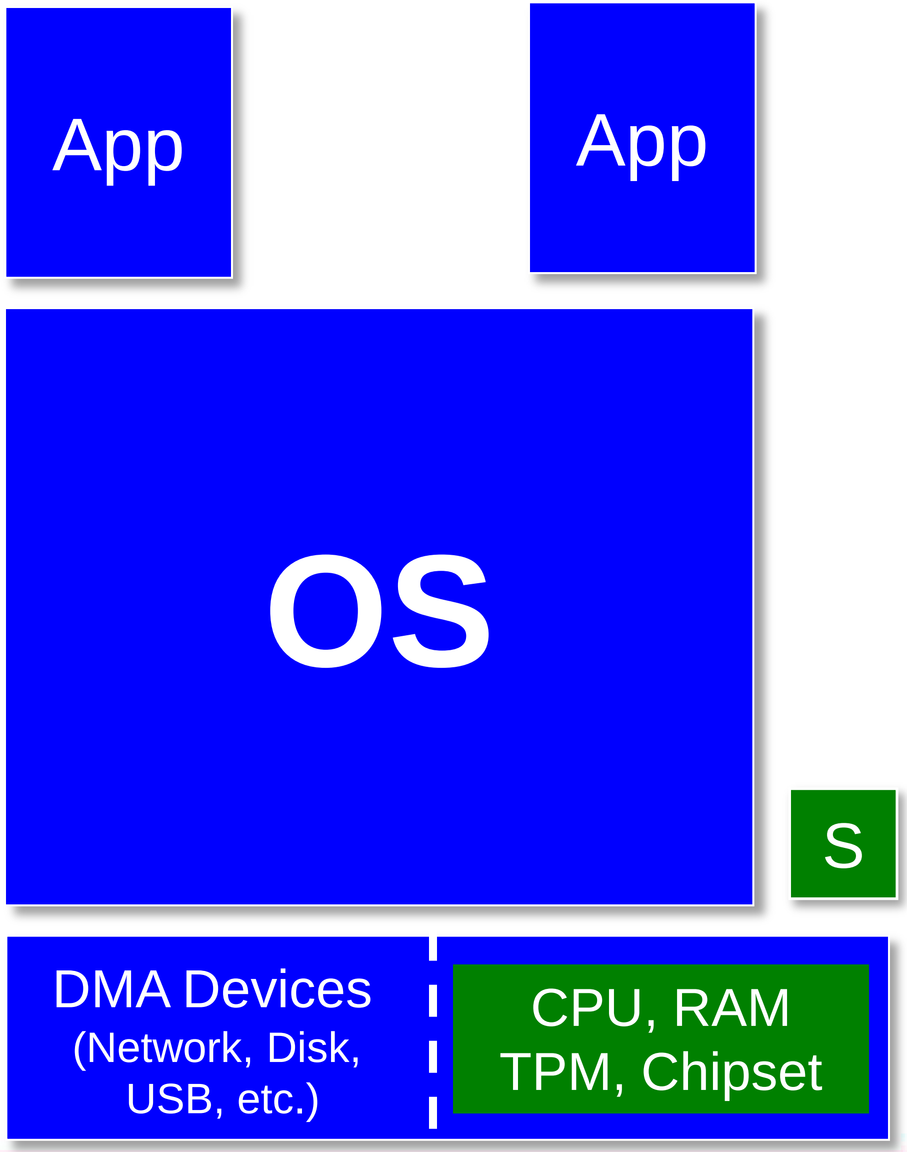

CyLab
Carnegie Mellon
Is my computer secure?
A

Goals

- Provide user with strong security properties
  - Execution integrity
  - Data secrecy and authenticity
  - Cyber-secure moments! © Virgil Gligor
- Compatibility with existing systems (both SW and HW)
- Efficient execution
- In the presence of malware
  - Assuming remote attacks: HW is trusted

Isolated Execution Environment (IEE)

- Execution environment that is defined by code S executing on a specific platform
  - Code is identified based on cryptographic hash H(S)
  - Platform is identified based on HW credentials
- IEE execution protected from any other code



Basic Trusted Computing Primitives

- Create isolated execution environment (IEE)
  - Create data that can only be accessed within isolated environment
- Remote verification of IEE
- Establish secure channel into IEE
- Externally verify that output O was generated by executing code S on input I protected by IEE

Basic Trusted Computing Primitives

- How to create IEE?
- How to remotely verify IEE?
- How to establish a secure channel into IEE?
- How to externally verify that output O is from S's computation on input I within IEE?

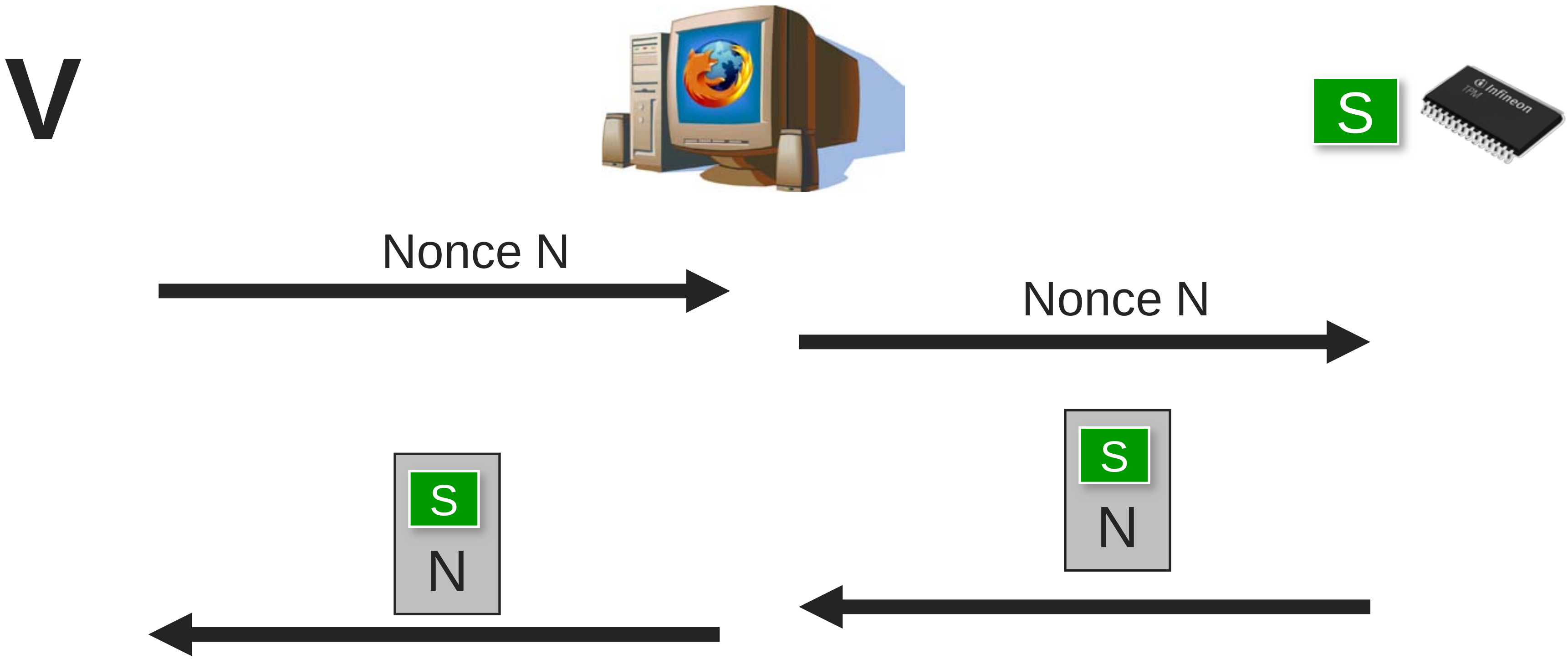
TPM Background

- The Trusted Computing Group (TCG) has created standards for a dedicated security chip: Trusted Platform Module (TPM)
- Contains a public/private keypair $\{K_{Pub}, K_{Priv}\}$
- Contains a certificate indicating that K_{Pub} belongs to a legitimate TPM
- Not tamper-resistant

How to Create IEE?

- AMD / Intel late launch extensions
- Secure Loader Block (SLB) to execute in IEE
- SKINIT / SENTER execute atomically
  - Sets CPU state similar to INIT (soft reset)
  - Enables DMA protection for entire 64 KB SLB
  - Sends [length bytes of] SLB contents to TPM
  - Begins executing at SLB's entry point

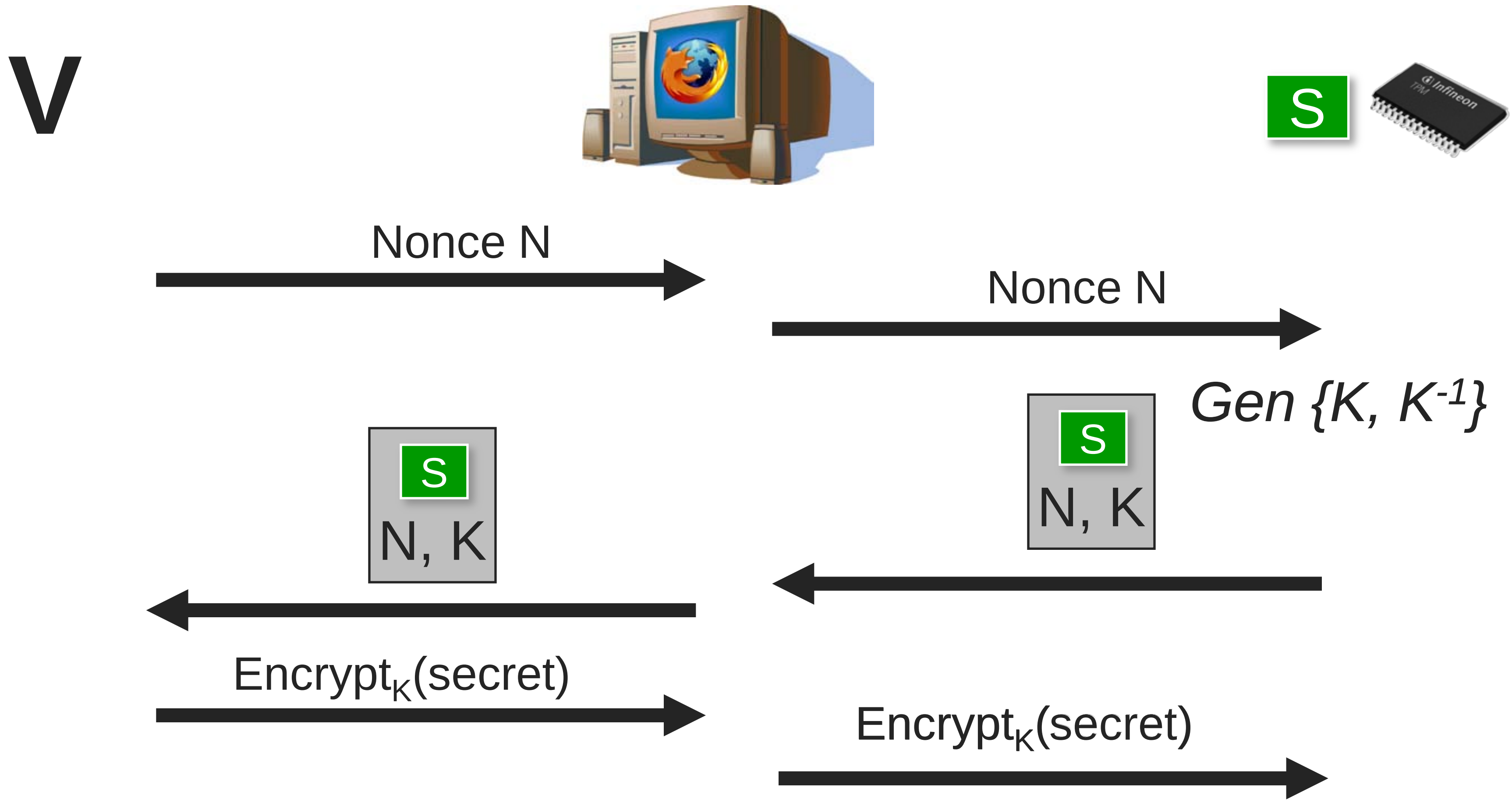
How to Remotely Verify IEE?



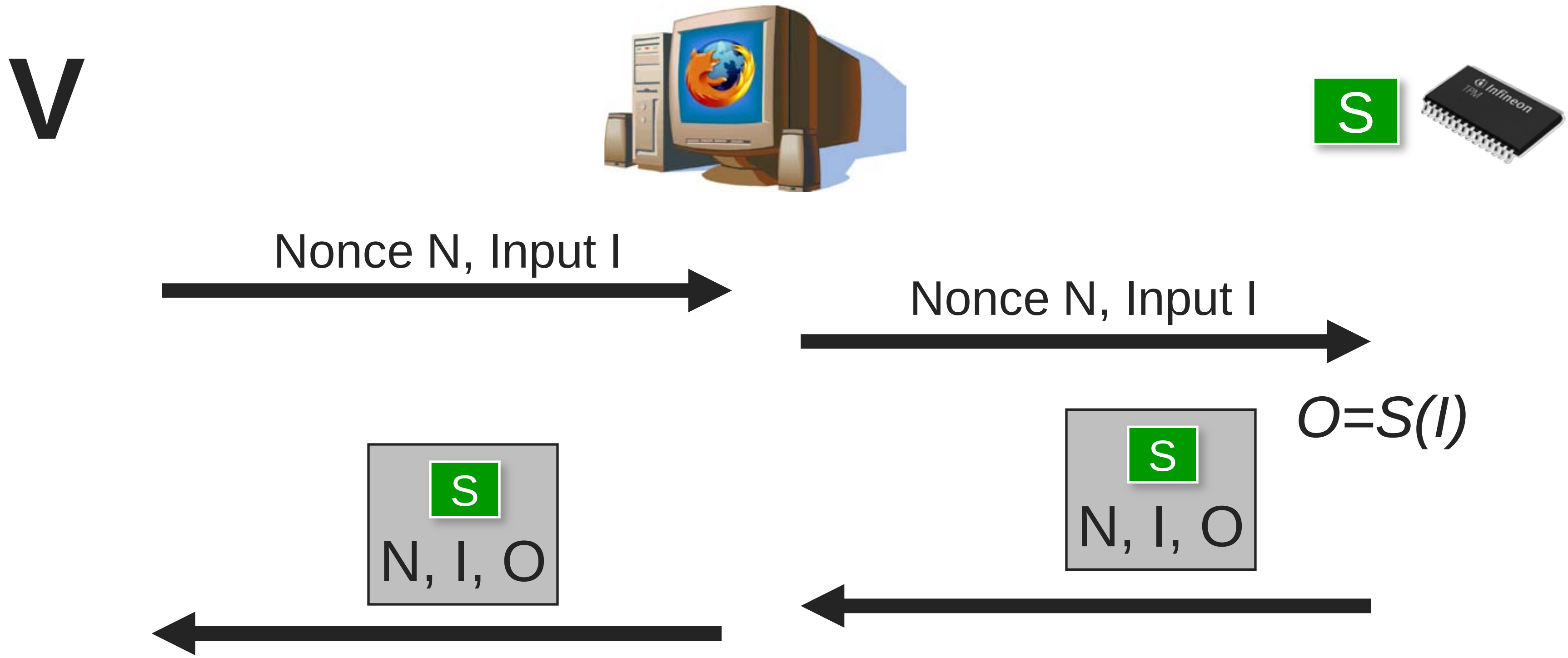
S
N

Means H(S) and N are signed by platform key

Secure Channel to IEE

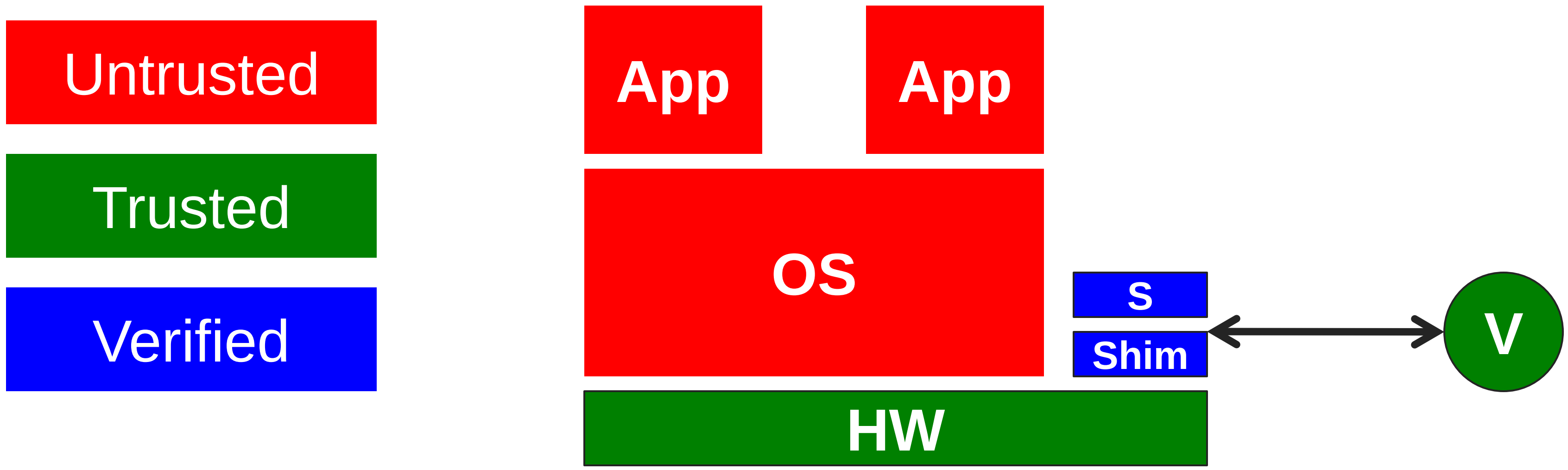


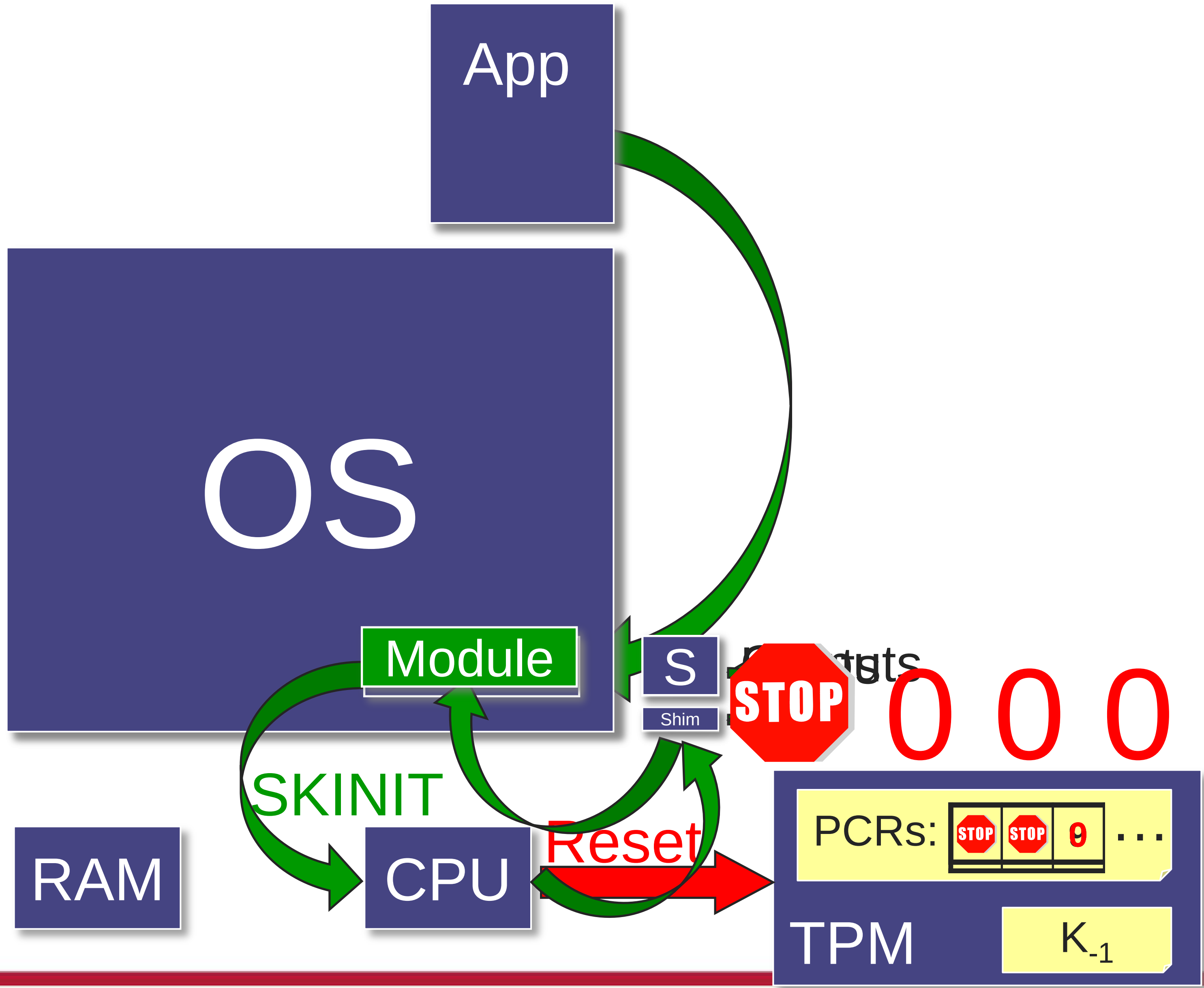
O=S(I) within IEE?

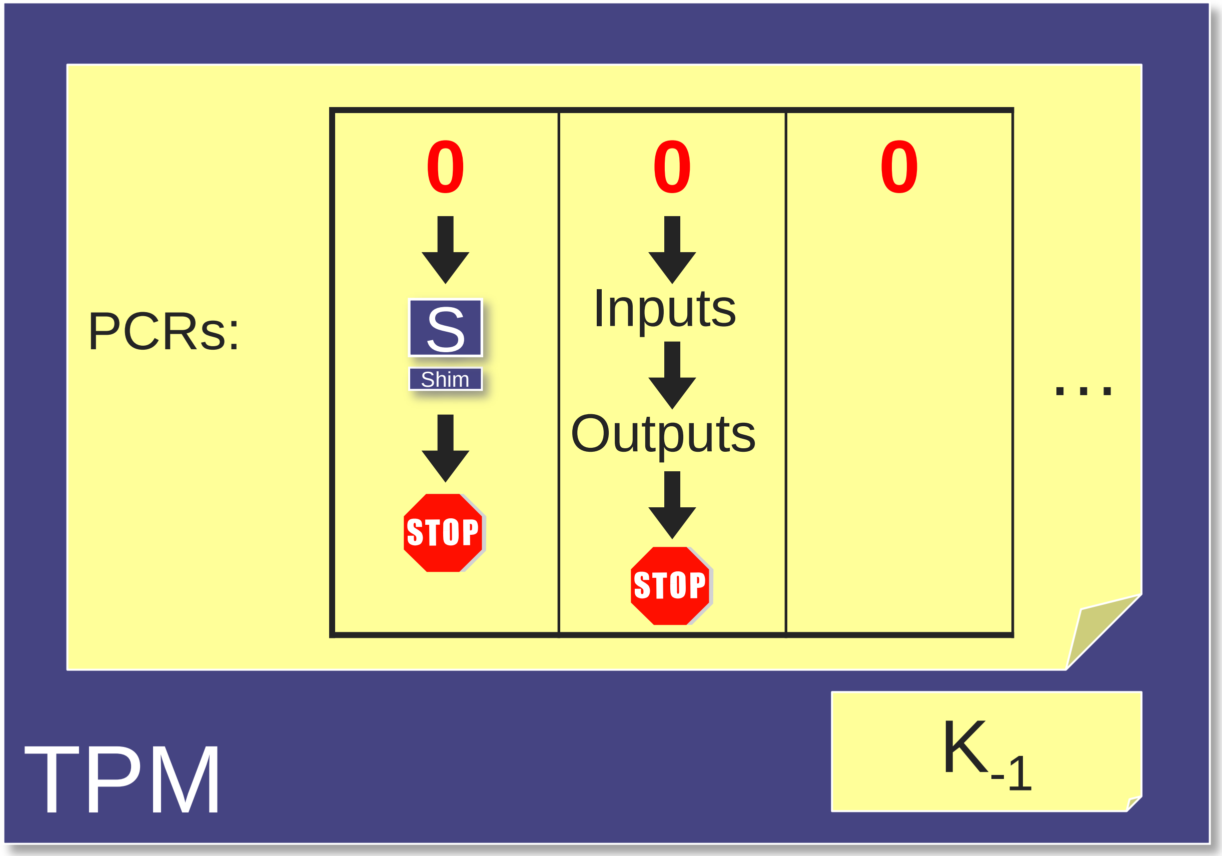


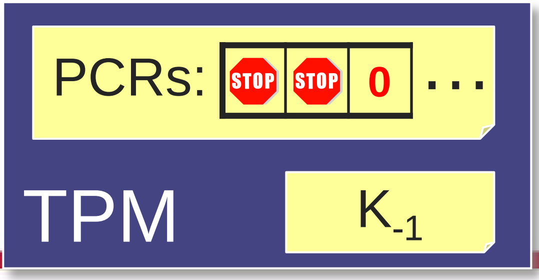
Flicker

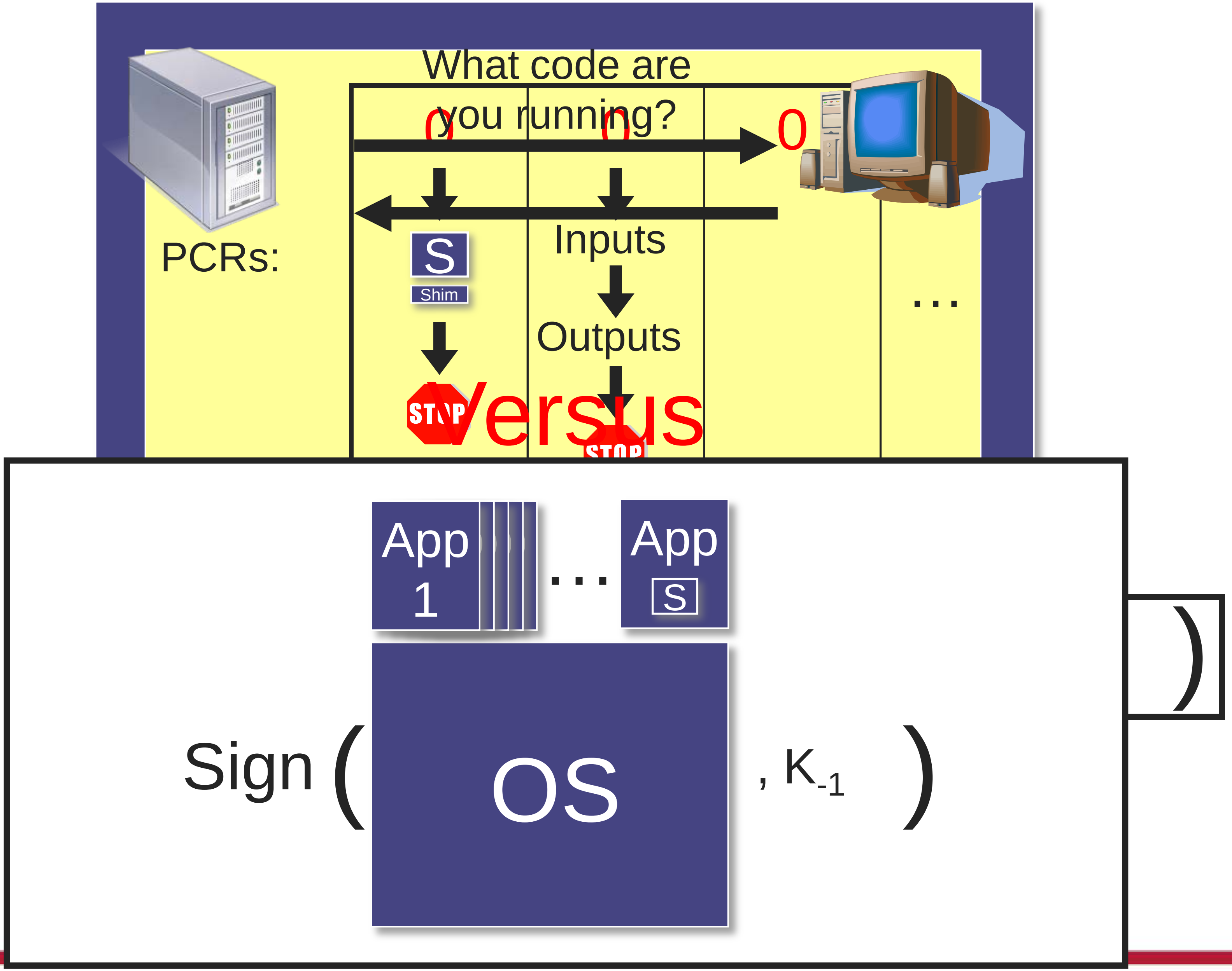
- McCune, Parno, Perrig, Reiter, and Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," EuroSys 08
- Goals
  - Isolated execution of security-sensitive code S
  - Attested execution of Output = S(Input)
  - Minimal TCB




App
OS
Module
S
Shim
STOP
0 0 0
SKINIT
Reset
RAM
CPU
PCRs:
STOP
STOP
0
...
TPM
K_{-1}


PCRs:
0
0
0
S
Shim
STOP
Inputs
Outputs
STOP
…
TPM
K_{-1}


PCRs:
STOP
STOP
0
…
TPM
K_{-1}

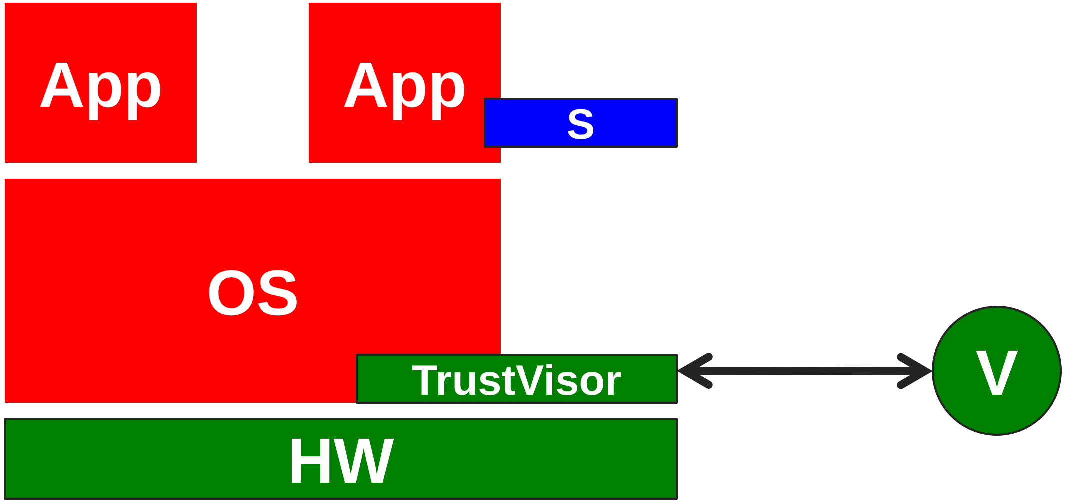

What code are you running?
PCRs:
0
0
0
S
Shim
Inputs
Outputs
STOP
Versus
STOP
...
App 1
...
App
S
OS
Sign (
, K_{-1}
)
)

Flicker Discussion

- Assumptions
  - Verifier has correct public keys
  - No hardware attacks
  - Isolated code has no vulnerabilities
- Observations
  - TCG-style trusted computing does not prevent local physical attacks
  - However, prevents remote attacks which are most frequent attacks

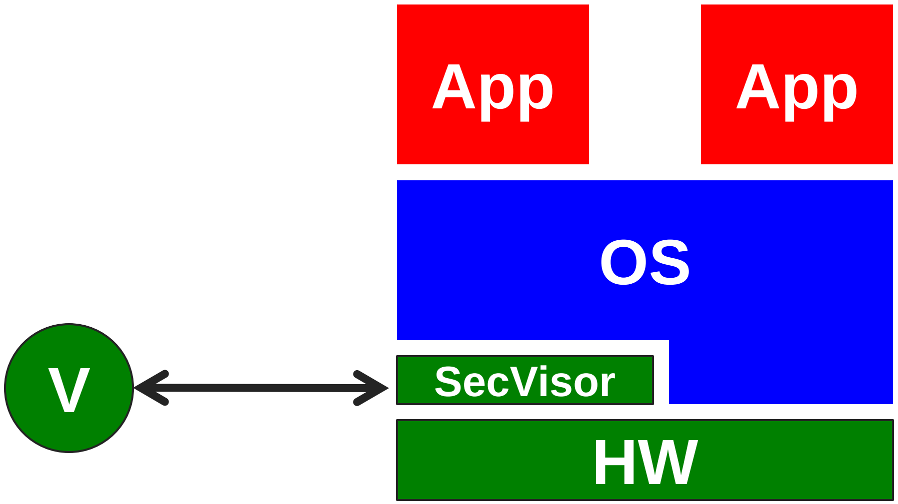
TrustVisor

- Goals
  - Similar to Flicker, replace min TCB by high efficiency
  - Isolated execution of security-sensitive code S
  - Attested execution of Output = S(Input)



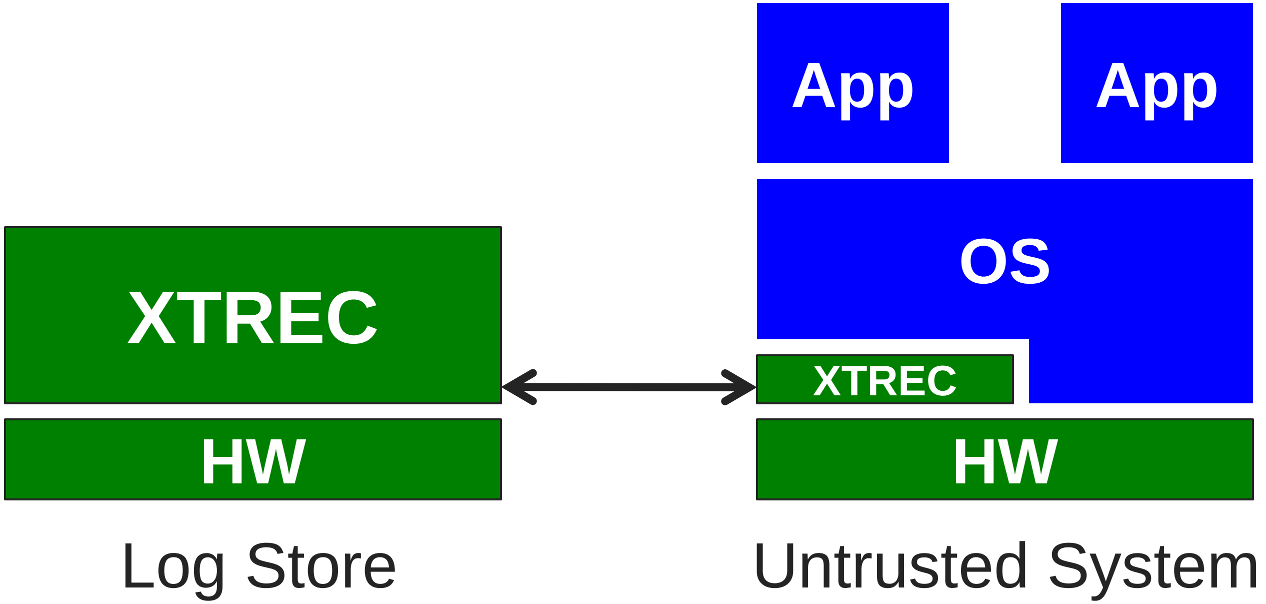
SecVisor

- Goals
  - Protect OS legacy against unauthorized writes
  - Code integrity property for untrusted OS: only approved code can execute in kernel mode
  - Attest to OS state to remote verifier



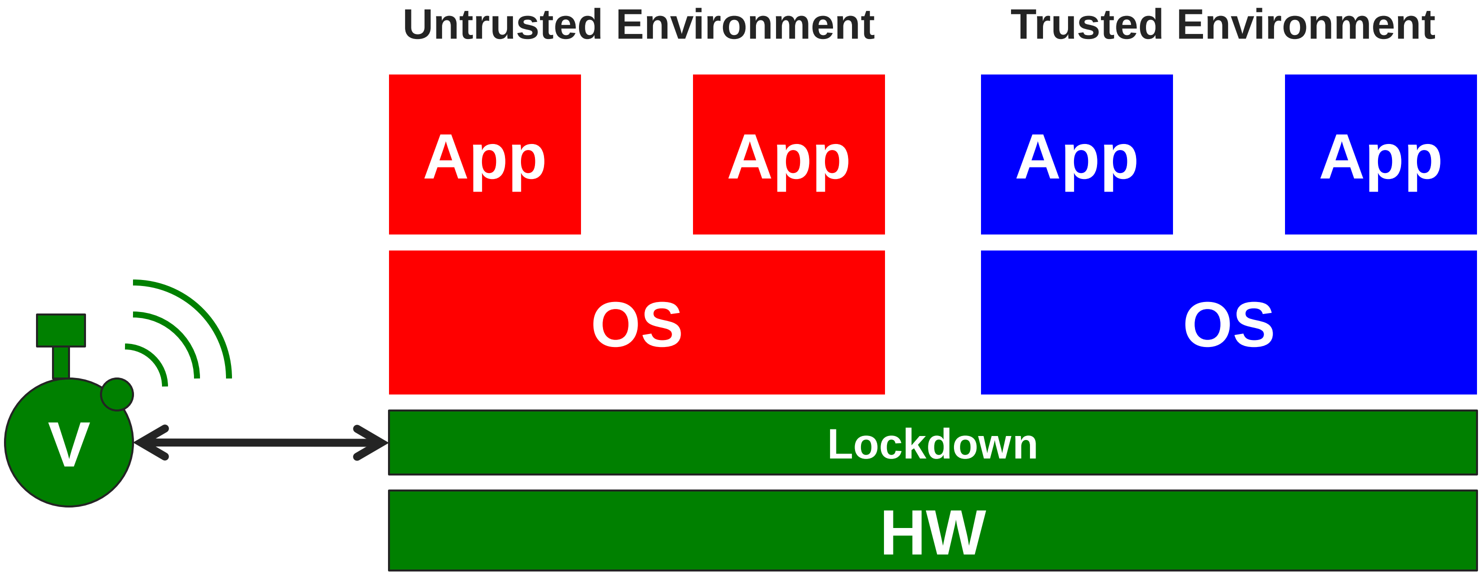
XTREC

- Goals
  - Complete execution tracing of a target system
  - Non-invasive, transparent
  - High performance



Lockdown

- Goals
  - Isolated execution of trusted OS environment
  - Trusted path to user
  - Protected secure browser in trusted OS



Conclusions

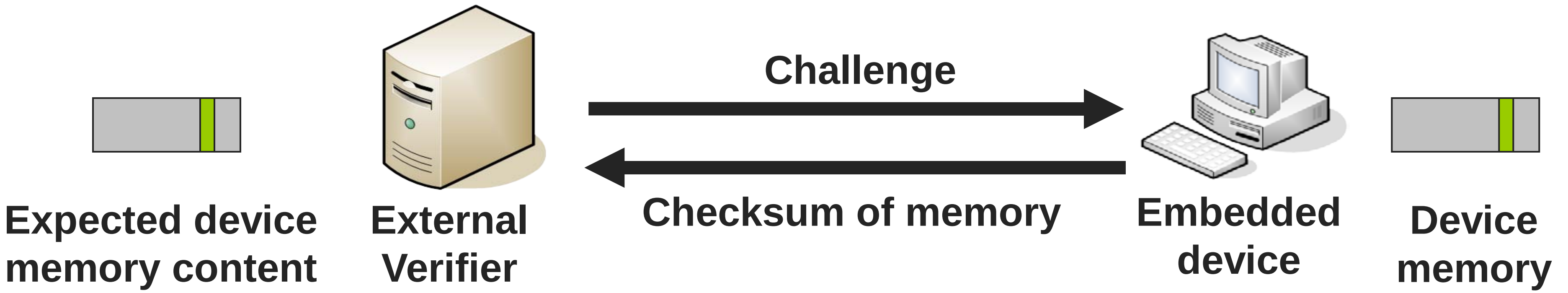
- Trusted computing mechanisms enable fundamentally new properties
  - On host: protect code & data even from admin
  - In distributed applications: simple data verification based on code that produced it
- Trusted computing mechanisms provide new primitives to build secure systems
- Trusted device can provide strong guarantees to local user

Software-Based Attestation

- Goal: provide attestation guarantees on legacy hardware, without trusted TPM chip
- Projects
  - SWATT: Software-based attestation, with Arvind Seshadri, Leendert van Doorn, and Pradeep Khosla [IEEE S&P 2004]
  - Pioneer: Untampered code execution on legacy hosts, with Arvind Seshadri, Mark Luk, Elaine Shi, Leendert van Doorn, and Pradeep Khosla [SOSP 2005]

Software-based Attestation Overview

- External, trusted verifier knows expected memory content of device
- Verifier sends challenge to untrusted device
  - Assumption: attacker has full control over device's memory before check
- Device returns memory checksum, assures verifier of memory correctness

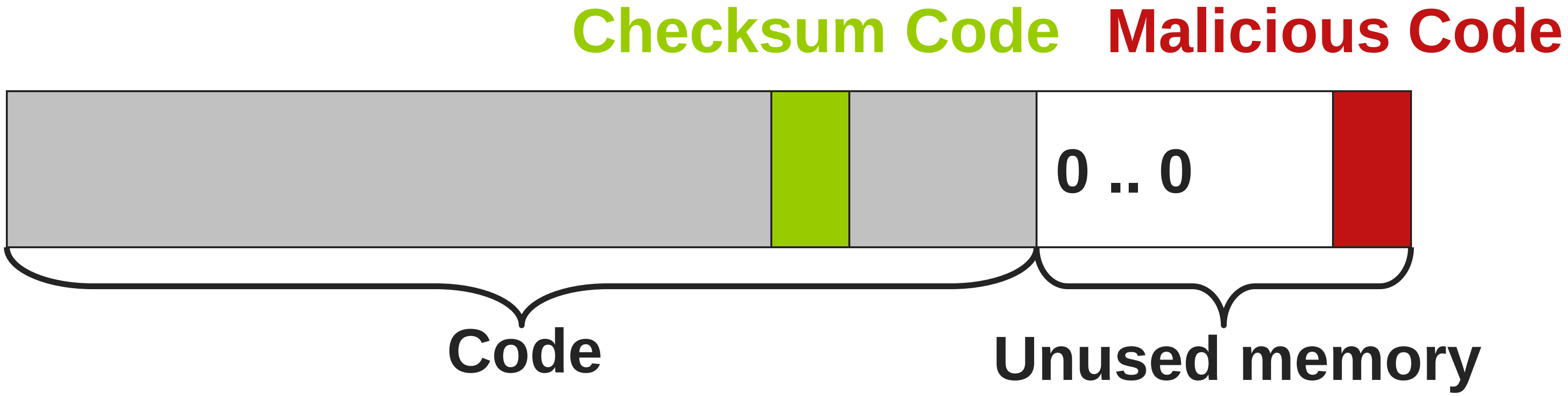


Assumptions and Attacker Model

- Assumptions on verifier
  - Knows hardware configuration of device
- Assumptions on device (untrusted host)
  - Hardware and firmware is trustworthy
  - Can only communicate with verifier: no proxy attacks
- Attacker controls device's software and OS before verification

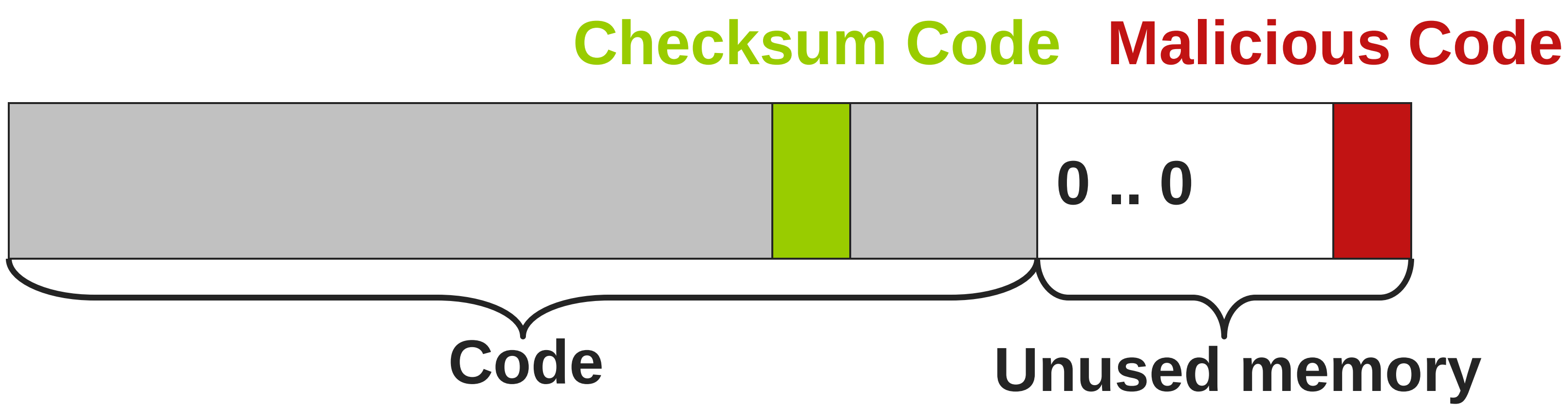
Checksum Function Design

- Approach 1: Verifier asks device to compute a cryptographic hash function over memory
  - V → D: Checksum request
  - D → V: SHA-1(Memory)
- **Attack**: malicious code pre-computes and replays correct hash value



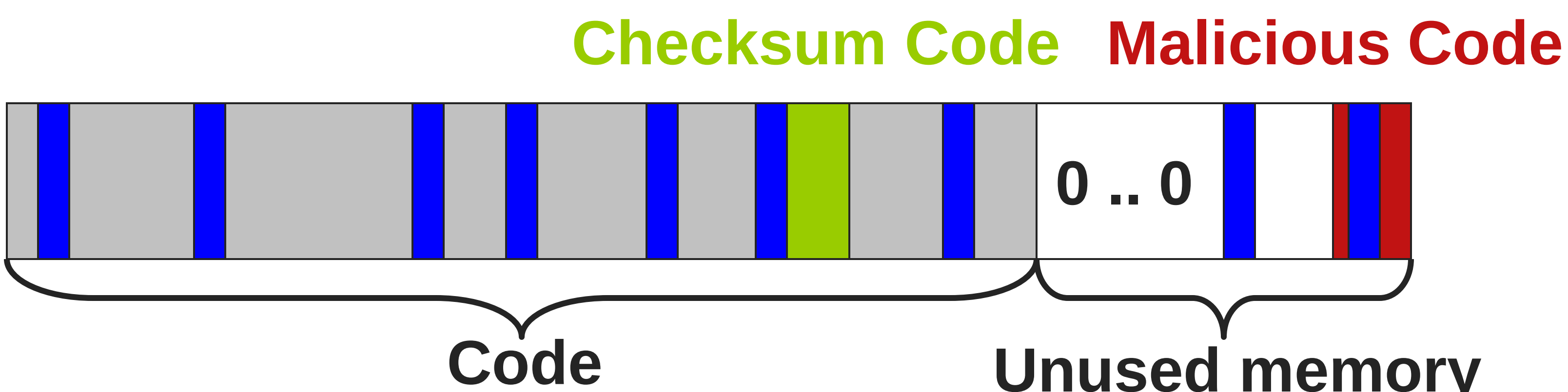
Checksum Function Design

- 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



Checksum Function Design

- Observation: need externally detectable property that reveals tampering of checksum computation
- Approach
  - Use time as externally detectable property, create checksum that slows down if tampering occurs
  - Compute checksum in pseudo-random order
  - Attacker needs to verify each memory access → slowdown

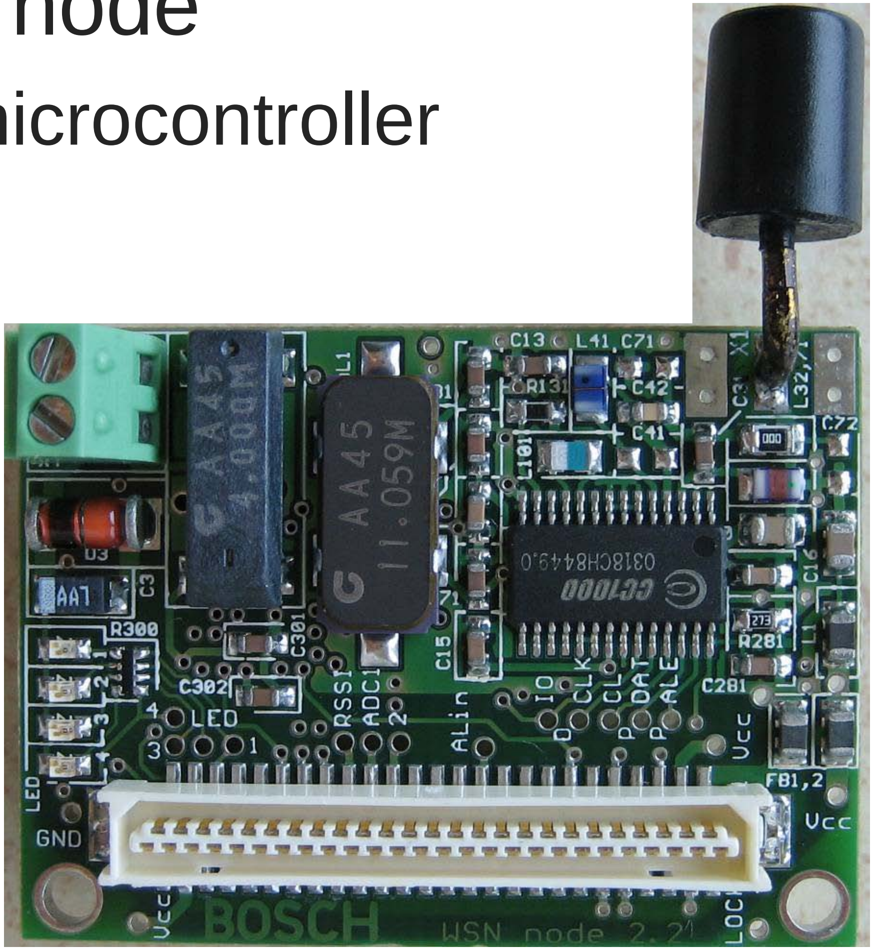


Checksum Requirements

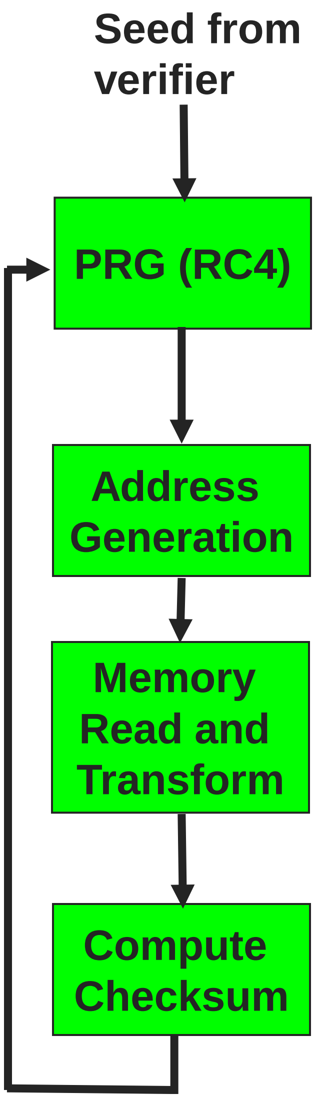
- Optimal implementation: code cannot be optimized
  - Denali project @ HP labs provides proof of optimal implementation of short pieces of code
  - GNU superopt
  - Open challenge to prove optimality of SWATT checksum
- No algebraic optimizations
  - Checksum has to be computed in entirety
  - Given a memory change, checksum cannot be "adjusted" without recomputation

Implementation Platform

- Bosch sensor node
  - TI MSP430 microcontroller



Assembler Code



Generate i^th member of random sequence using RC4

```
zh  = 2               ldi zh, 0x02
r15 = *(x++)          ld r15, x+
yl  = yl + r15        add yl, r15
zl  = *y              ld zl, y
*y  = r15             st y, r15
*x  = r16             st x, r16
zl  = zl + r15        add zl, r15
zh  = *z              ld zh, z
```

Generate 16-bit memory address

```
zl  = r6              mov zl, r6
```

Load byte from memory and compute transformation

```
r0  = *z              lpm r0, z
r0  = r0 xor r13      xor r0, r13
r0  = r0 + r4         add r0, r4
```

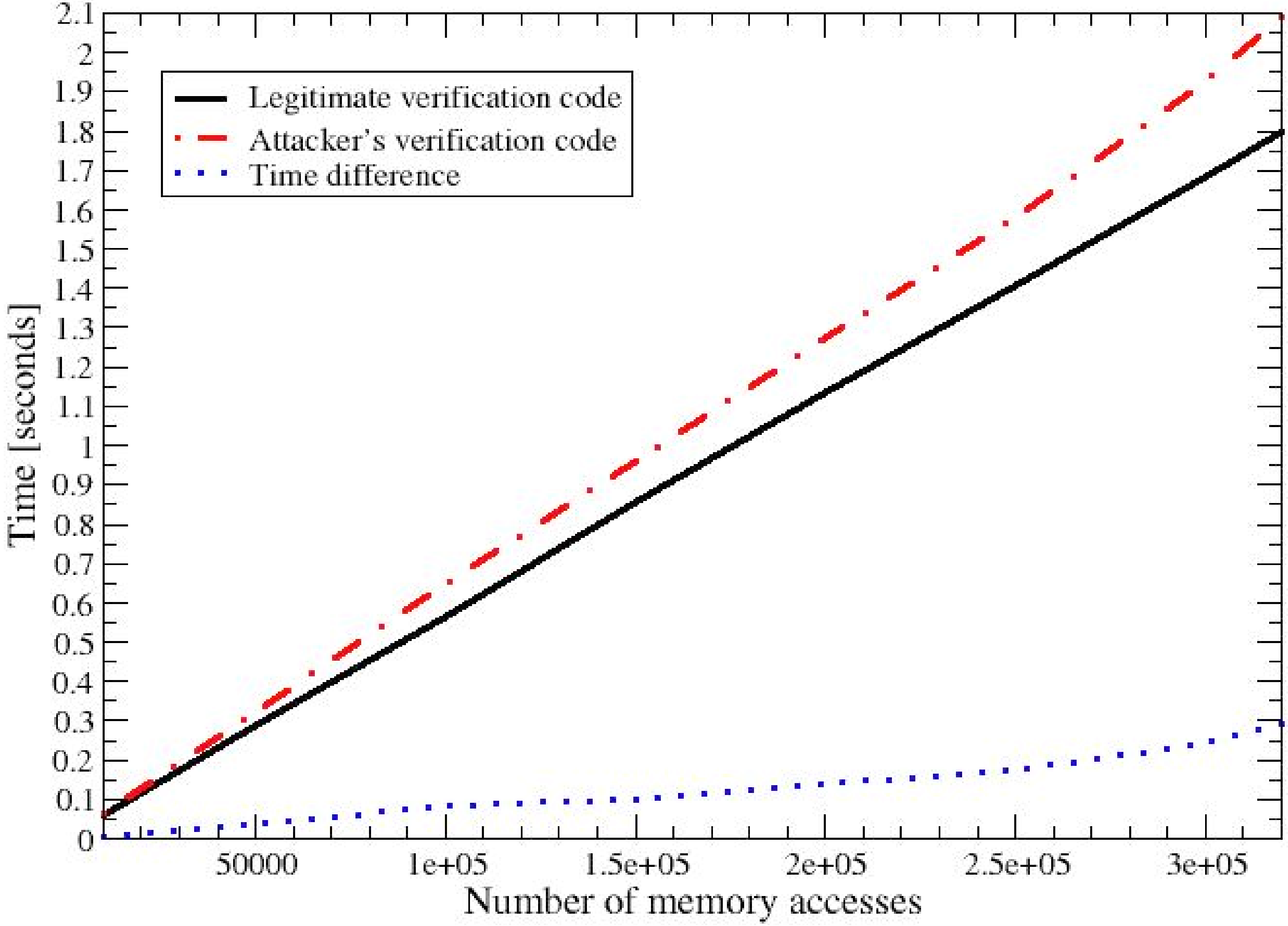
Incorporate output of hash into checksum

```
r7  = r7 + r0         add r7, r0
r7  = r7 << 1         lsl r7
r7  = r7 + carry_bit  adc r7, r5
r4  = zh              mov r4, zh
```

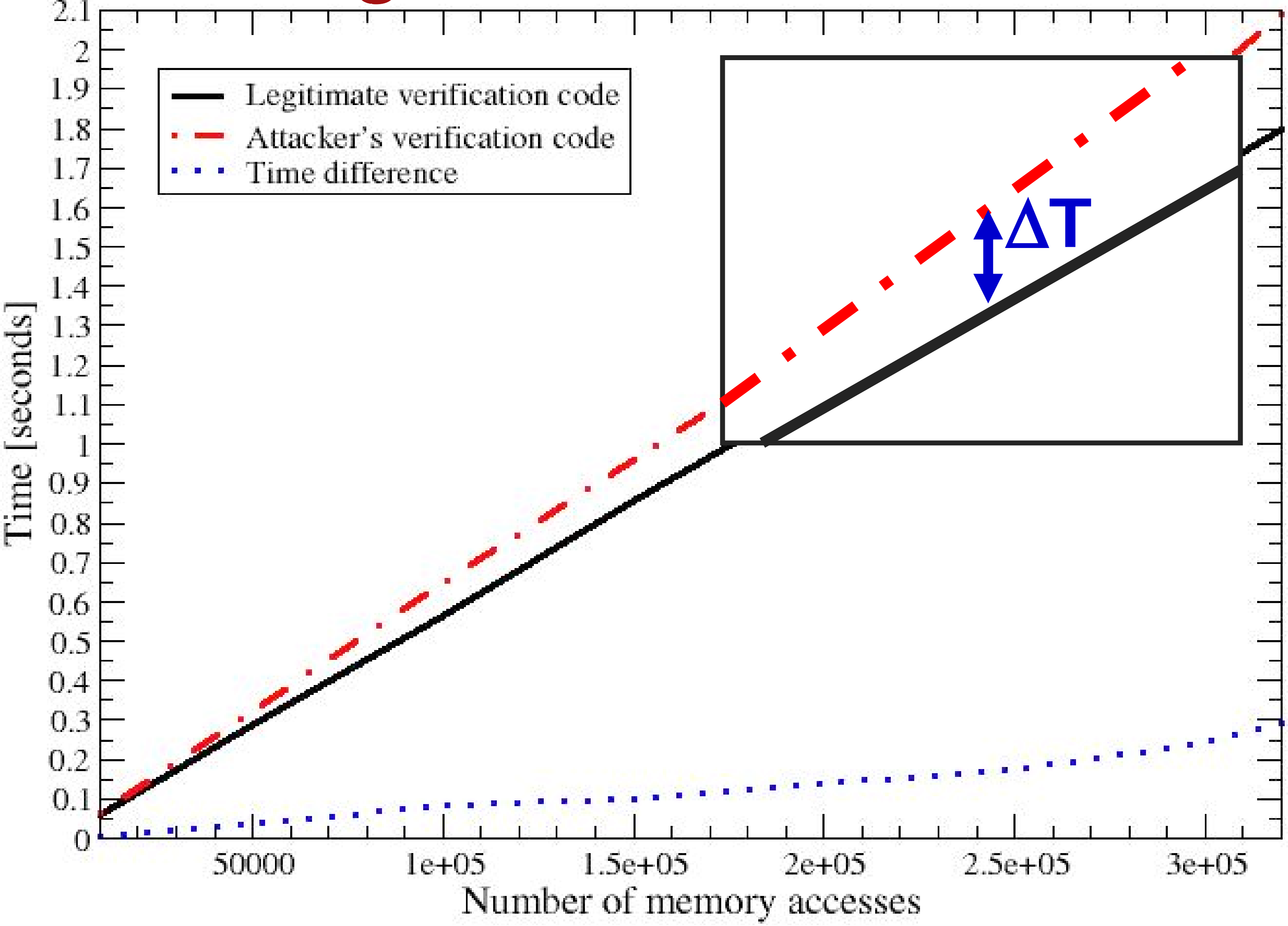
SWATT Advantage

- 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

Results


Legitimate verification code
Attacker's verification code
Time difference
Time [seconds]
Number of memory accesses
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2
2.1
50000
1e+05
1.5e+05
2e+05
2.5e+05
3e+05

Selecting Number of Iterations

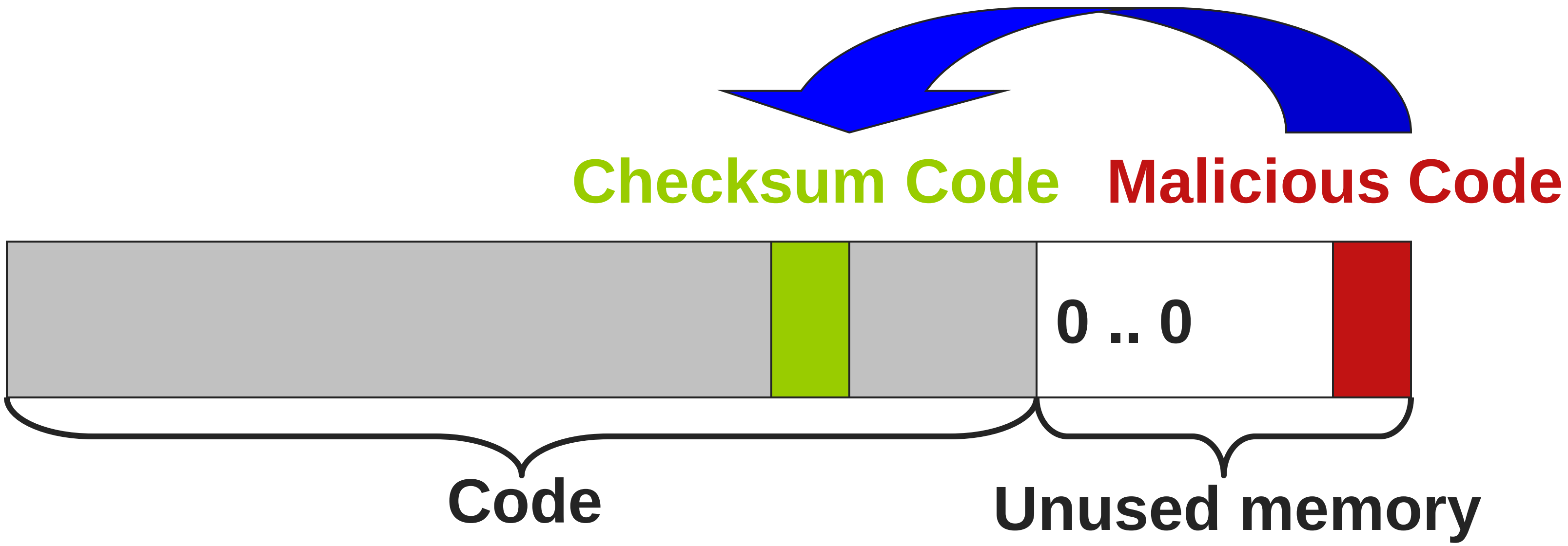


SWATT Extension

- 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!

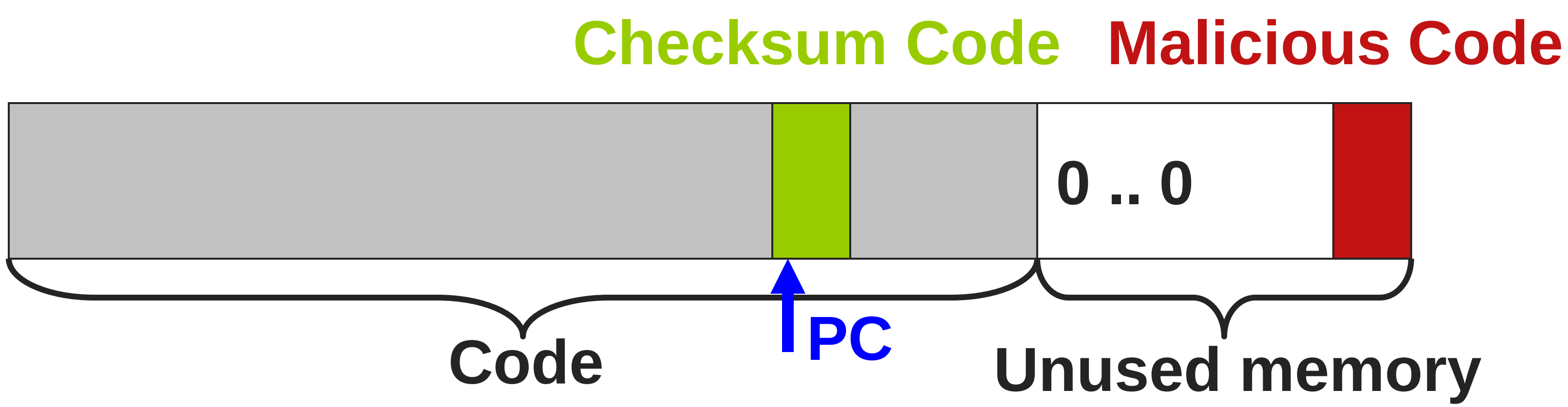
Attack on Partial Memory Verification

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

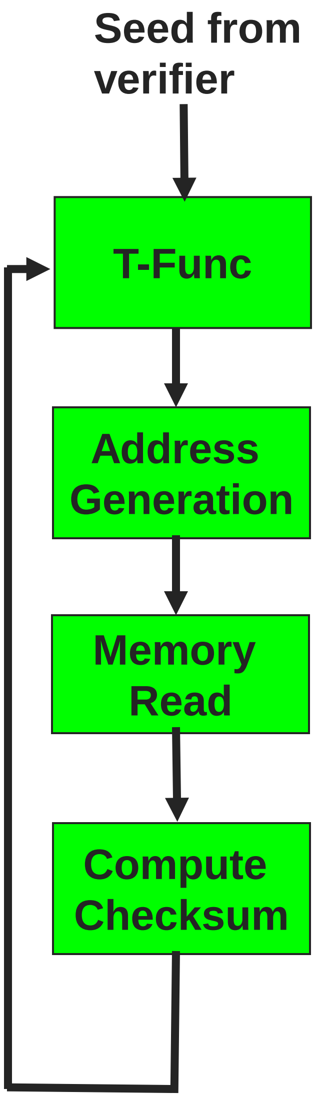


Improved Checksum Approach

- 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 Assembler 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

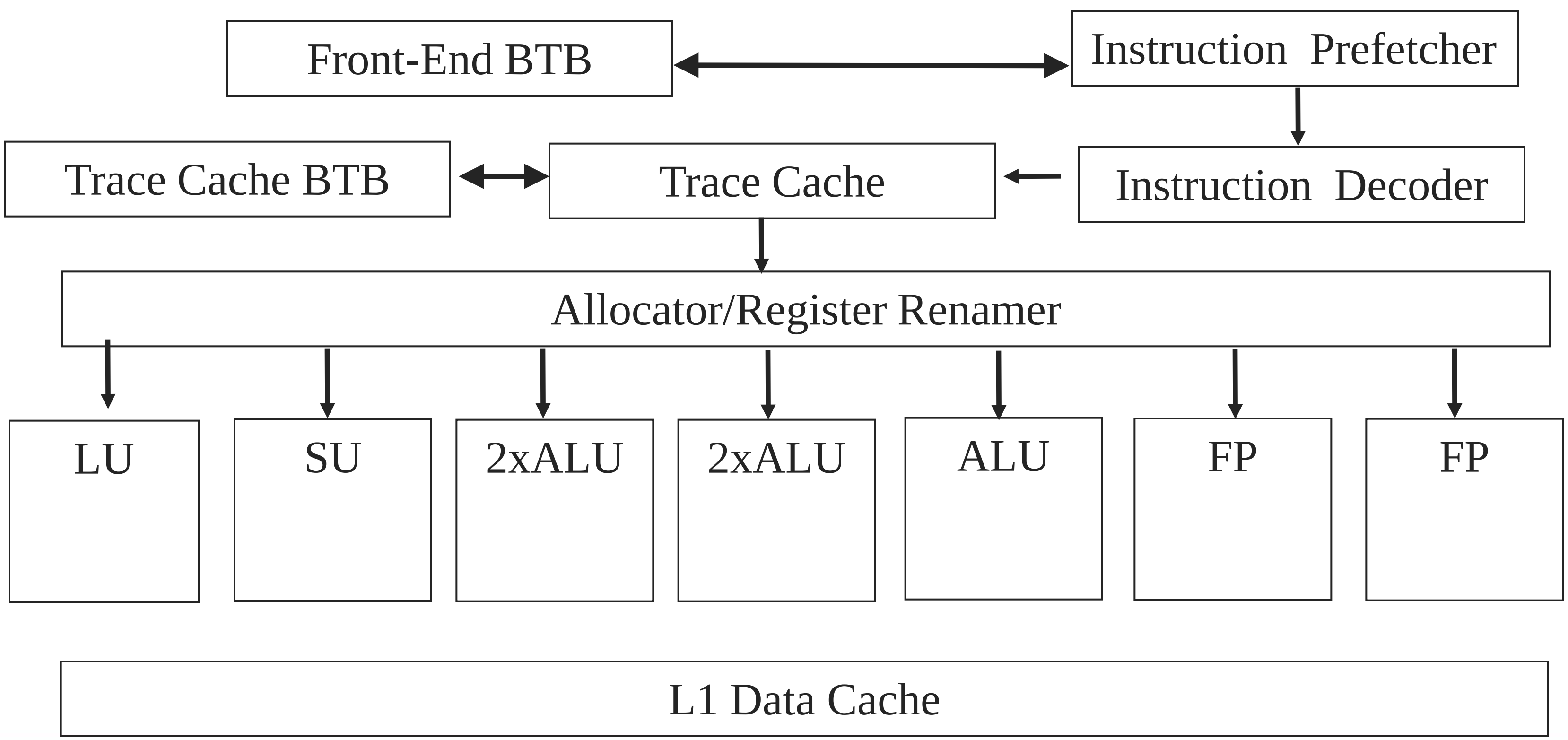
- First step to address untampered code execution on untrusted legacy hosts
- Implemented on Intel Pentium IV
  - Numerous challenges exist on this platform!
- Designed a kernel rootkit detector using Pioneer, to guarantee that correct code has executed on untrusted host

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

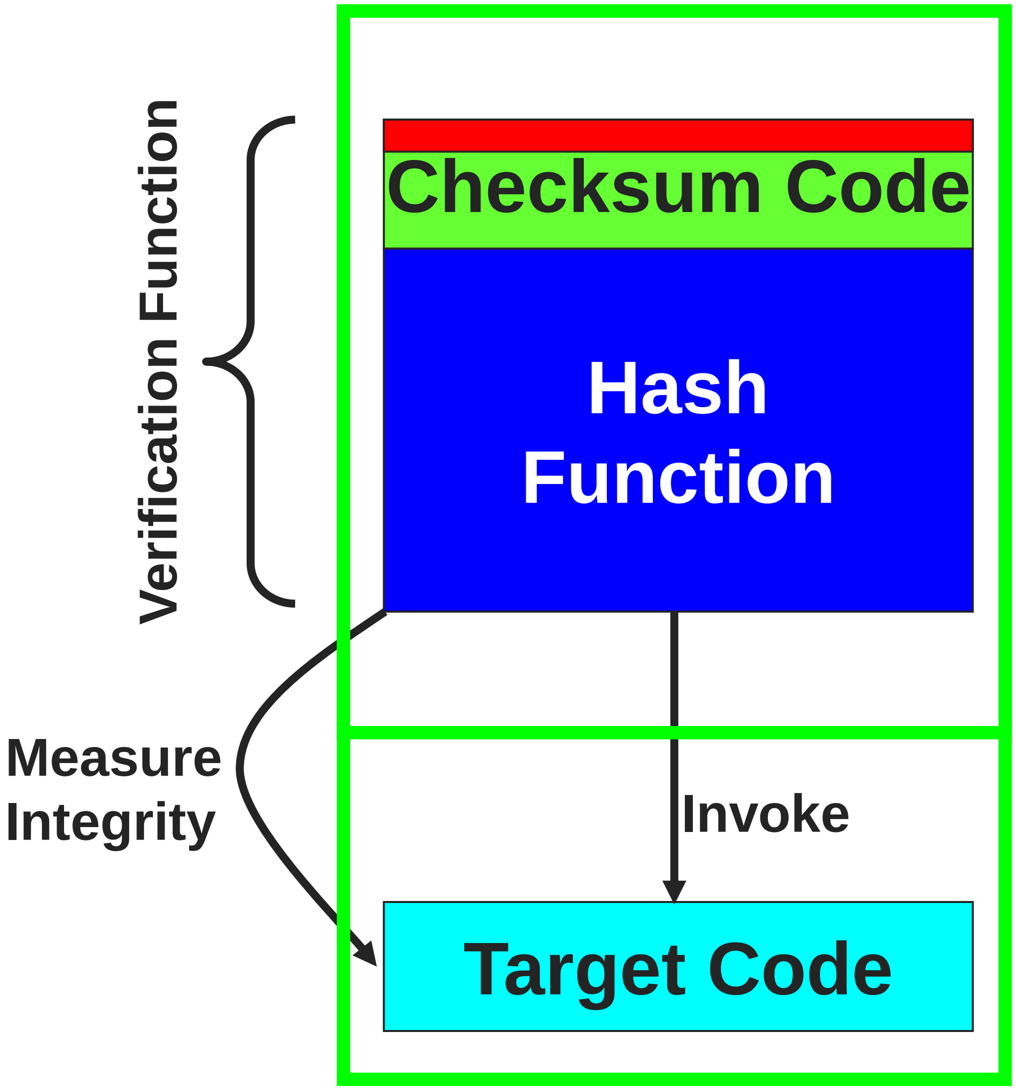
- Intel Xeon @ 2.8 GHz, Linux kernel 2.6.7
  - Intel Netburst Microarchitecture (Pentium 4)
  - Key: issue max 3 μops per cycle (3 way superscalar)
  - 64-bit extensions (no segmentation)



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

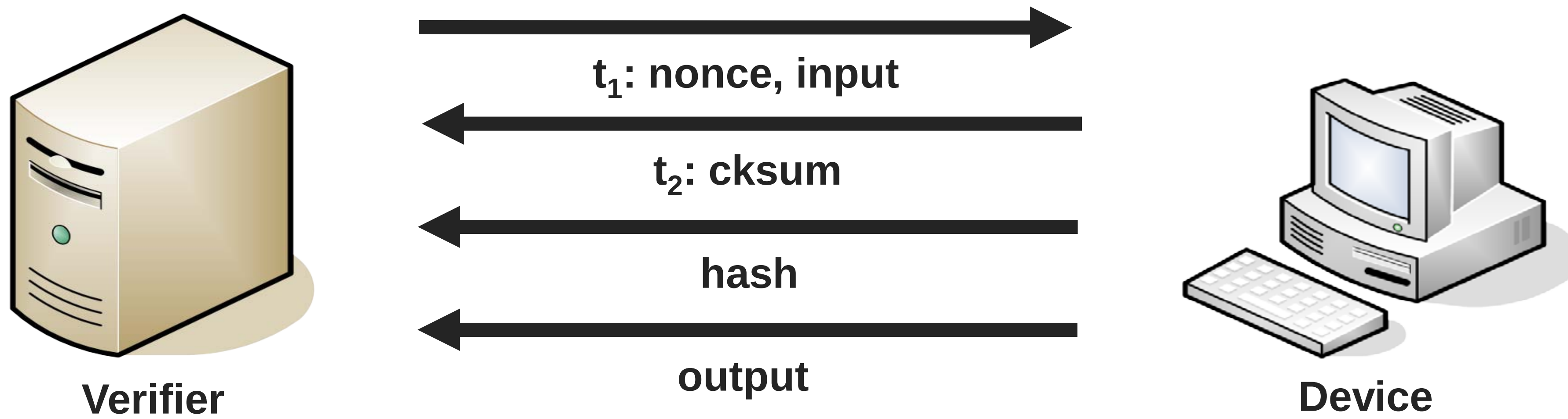
Design of Verification Function



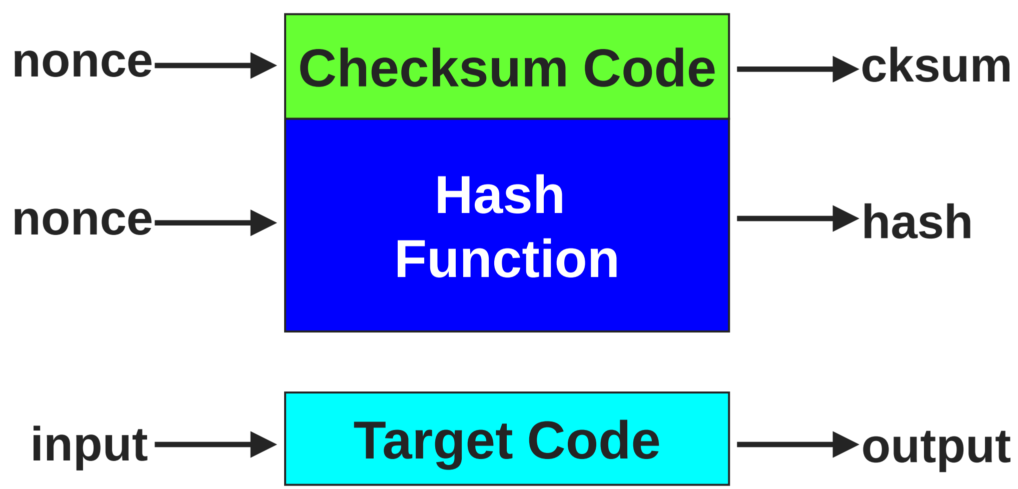
Root of Trust

- Compute checksum
- Set up untampered execution environment

The Pioneer Protocol



- **Successful verification if:**
  $t_2 - t_1$ < expected time &&
  cksum == exp. cksum



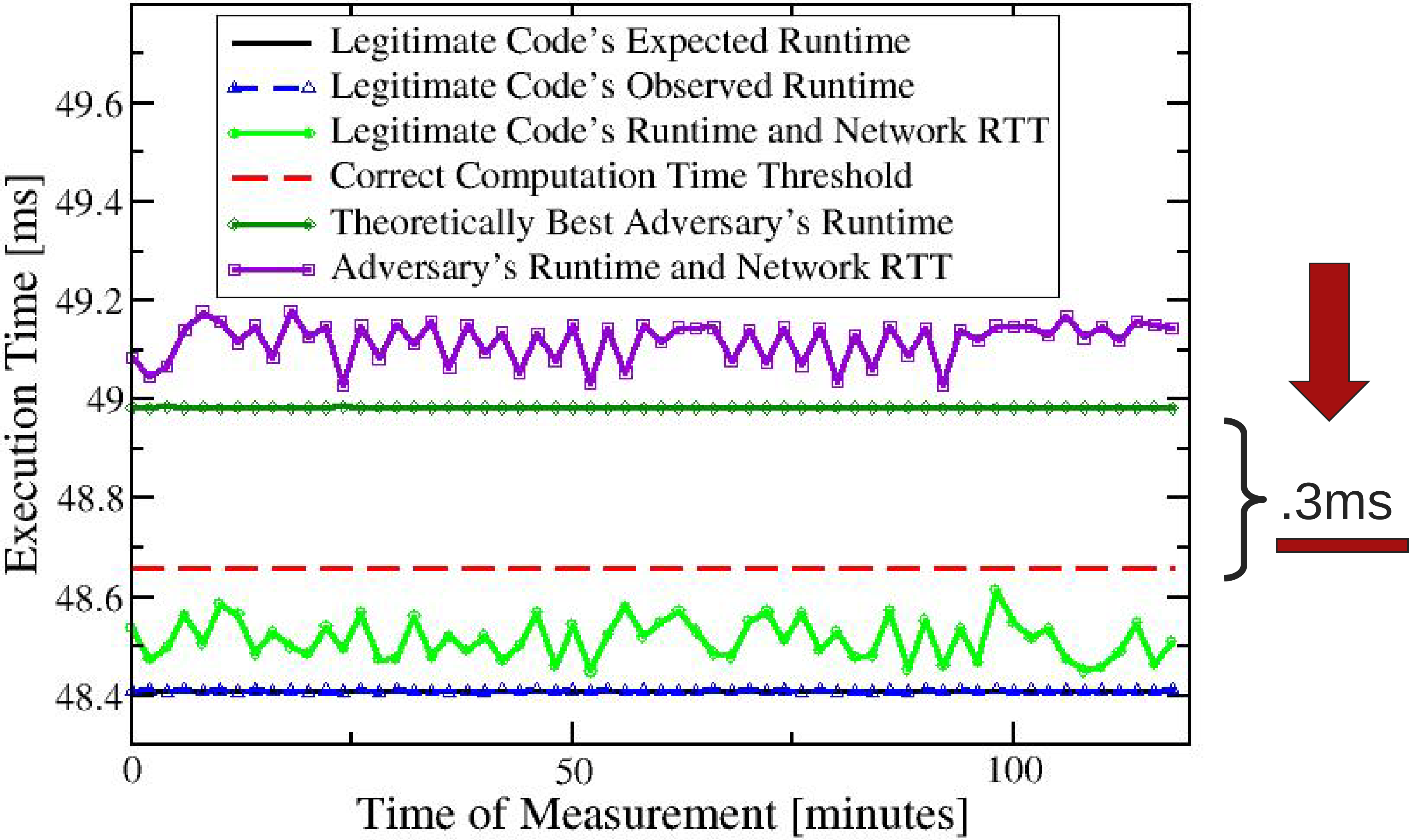
Desired Security Property

- 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

Results – Runtime Difference



Pioneer Discussion

- Verifier can obtain untampered execution guarantee for code executing on untrusted platform
- Similar attestation property to AMD SVM or Intel TXT
- Drawback: Requires defense against proxy attack