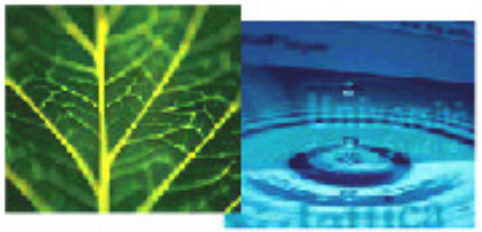



Swatt: Software-based Attestation for Embedded Devices[1]

Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems[2]

Mariano Ceccato

ceccato@itc.it

[1]SWATT: softWare-based attestation for embedded devices Seshadri, A.; Perrig, A.; van Doorn, L.; Khosla, P. Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on, Vol., Iss., 9-12 May 2004 Pages: 272- 282

[2]Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. 2005. “Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems”. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 23 - 26, 2005). SOSP '05. ACM Press, New York, NY, 1-16.

Outline

- Verifiable code execution.
- TPM approach.
- Swatt/Pioneer approach.
- Swatt architecture.
- Pioneer architecture.
- Adoption scenarios.

The problem

Verifiable code execution:

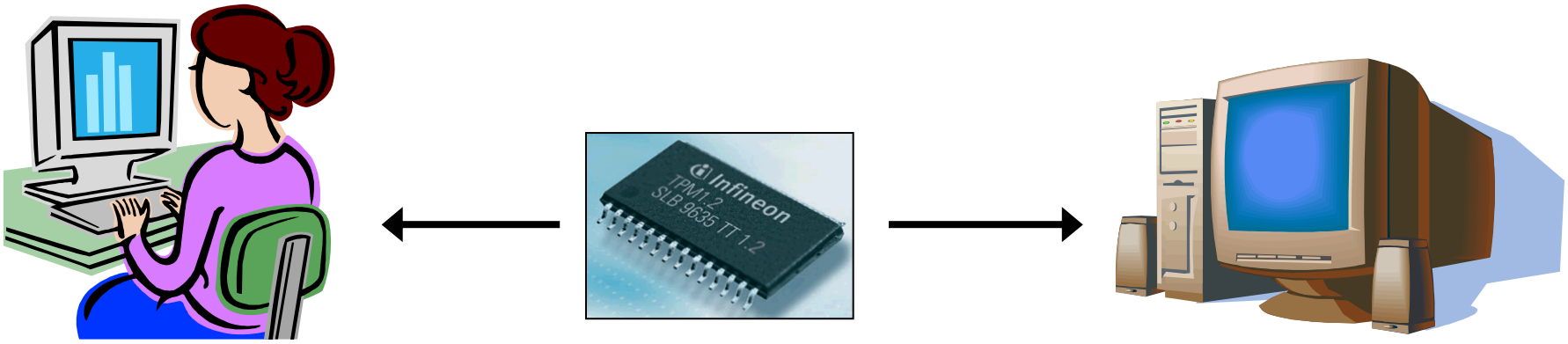
- Verifying that some arbitrary code is executed un-tampered on an un-trusted platform, even in the presence of malicious software on that platform.
  - The code is not modified before being invoked.
  - No alternate code is executed.
  - The execution state is not modified at run-time.

Adoption scenarios:

- Integrity check of network printers.
- Virus presence on smart cell phones equipped with e-mail client.
- Software certification on e-voting machines.
- Presence of root-kit in OS-kernel.

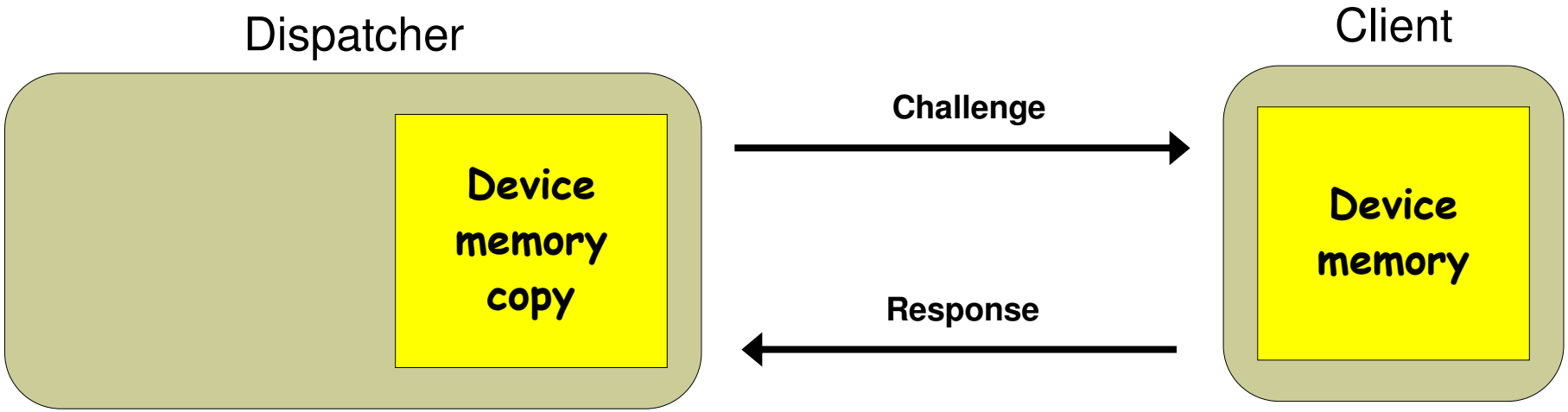
Hardware based attestation

- TPM is used to measure the state of the platform during the boot process.
- Malicious code is detected because it causes measurements to deviate from the expected values.
- Measurements are stored in the Platform Configuration Registers (PCR) within TPM.
- Remote attestation allows a party to obtain assurance in the correct operation of a remote system.



Swatt & Pioneer

- Software based primitive to verify code execution on an un-trusted host
  - It can be updated.
  - No special purpose hardware is required.
  - No particular CPU extension (*e.g.,* virtualization).
  - It provides run-time attestation.
- It is based on
  - Challenge-response protocol.
  - External trusted entity, which can not be compromised by an attacker.
  - Communication link.



Assumptions

Dispatcher:

- It knows the exact hardware configuration of the un-trusted client
  - Memory size and architecture.
  - Instruction set architecture.
  - Processor clock speed.

Communication channel:

- Message origin authentication.
- Un-trusted platform can only communicate with the dispatcher when the verifier runs.

Attacker model

- The attacker could have the complete control of the software on the un-trusted platform (administrator privileges)
  - Applications.
  - Operating system.
  - Direct access to the memory.

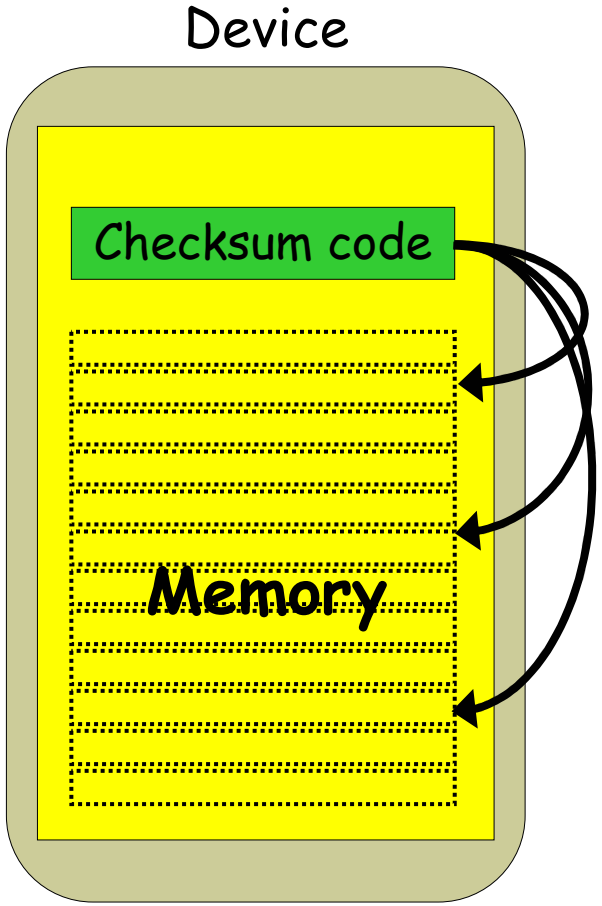
- The attacker can not modify the hardware
  - He can not load malicious firmware on disk controllers or network interfaces.
  - He can not attach more memory to the system.
  - He can not replace the processor with a faster one.

SWATT

- Attestation of embedded devices:
  - Cell phones, PDA
  - Network printers
  - Micro-controllers (fire detector, climate monitoring)
  - Sensor network
- It relies on an external verifier, because without secure hardware a potentially compromised device can not verify itself correctly.

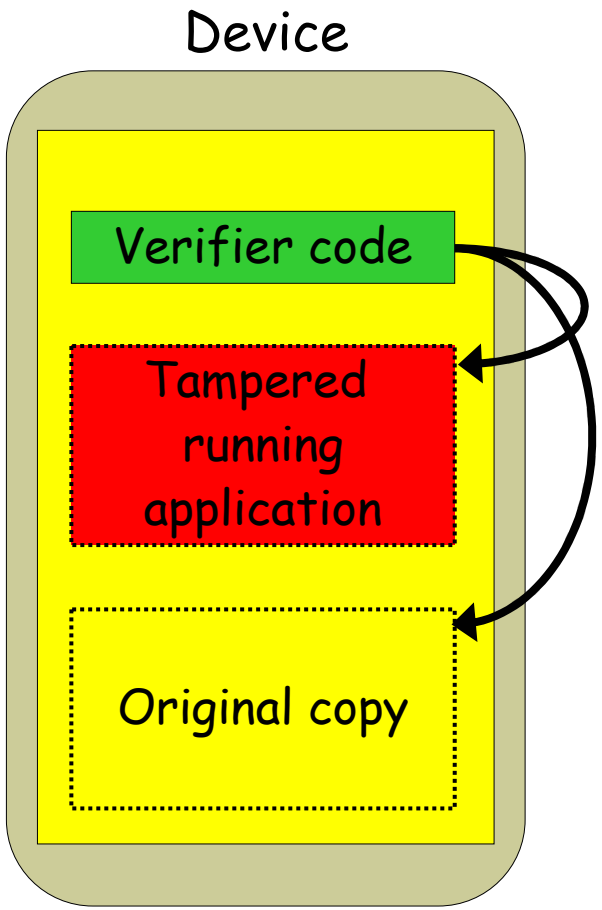
Verification function

- Based on Message Authentication Code (MAC) of the memory contents.
- Random key sent as challenge prevent MAC pre-computation attacks.
- Random memory traversal to touch with high probability all the memory locations.
- Optimized implementation and non parallelizable.



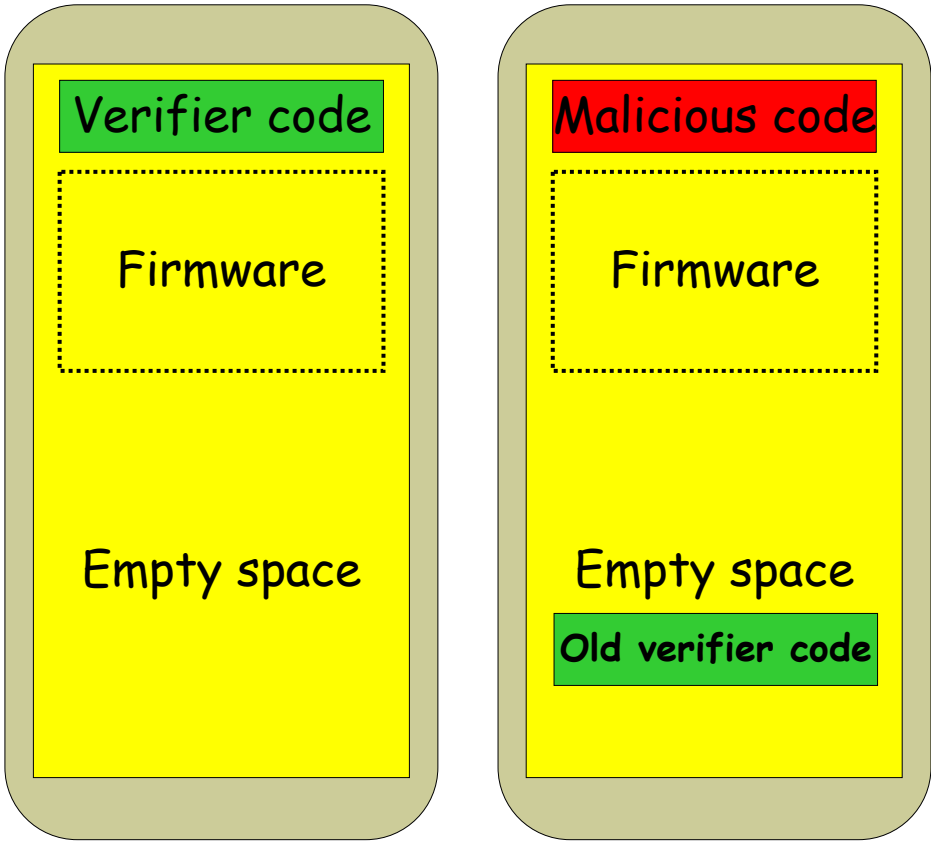
Memory copy attack

- Small implementation to prevent memory copy attack
  - The main loop takes 23 machine cycles.
  - An attack consists in a comparison and a conditional jump
    - 3 machine cycles
    - +13%.
  - Execution time is predictable.



Memory verification attack

- Verifier is replaced by malicious verification code.
- Old verification code is copied into empty memory.
- Attacker saves time when a (0 filled) empty memory region is check-summed by skipping the computation of the checksum.

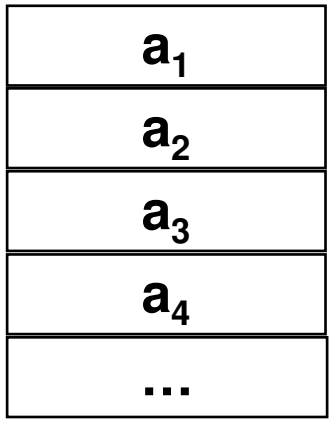


Expected memory layout

Attacker's memory layout

Checksum function

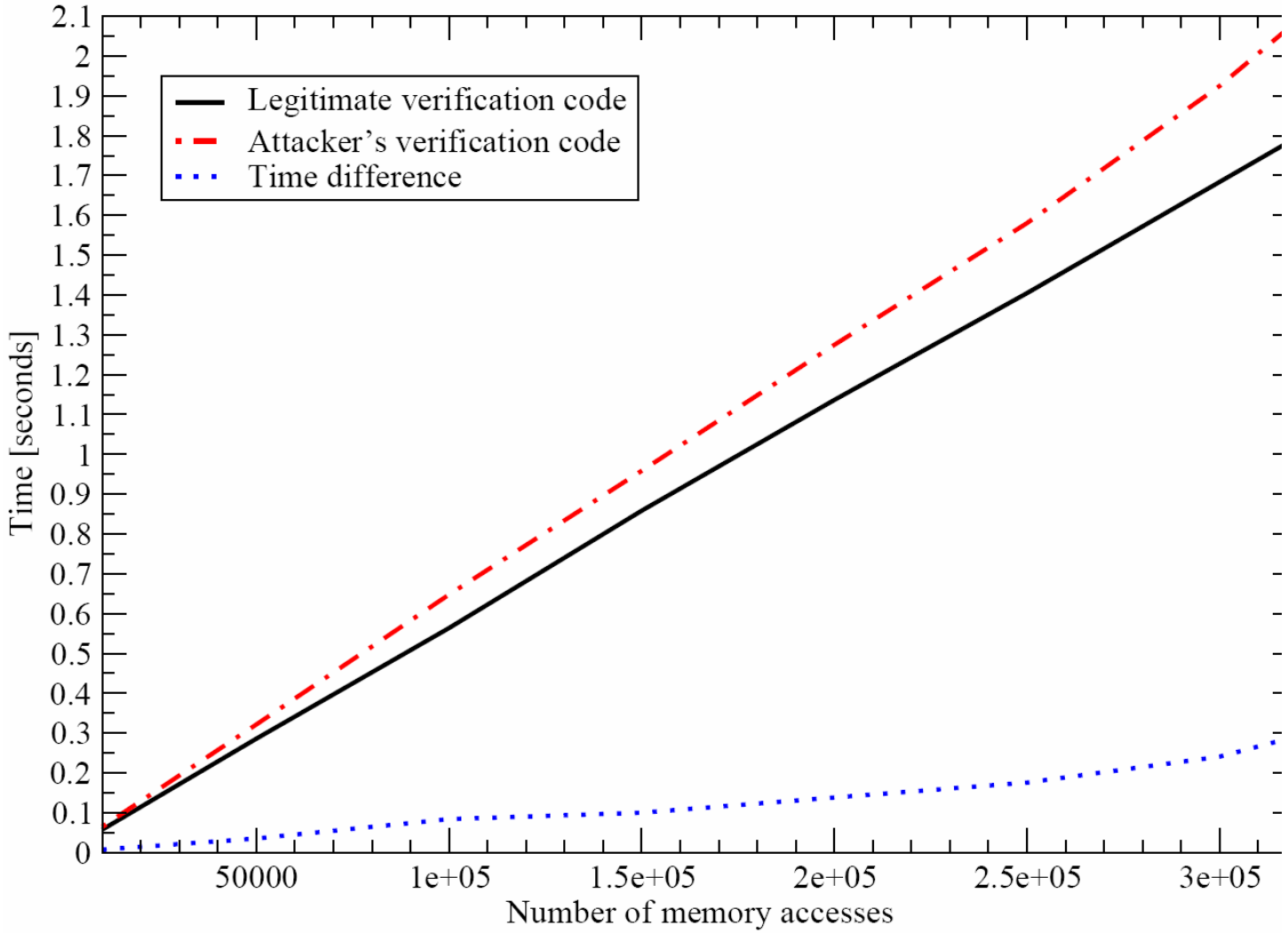
- A tampered checksum computation results in time overhead.
- The adversary could use saved time to forge the checksum.
- Function implemented as sequence of XOR and AND.
  - Difficult to parallelize.
  - Strongly ordered.
  - Optimal implementation.



$$checksum = [(a_1 \oplus a_2) + a_3] \oplus a_4$$

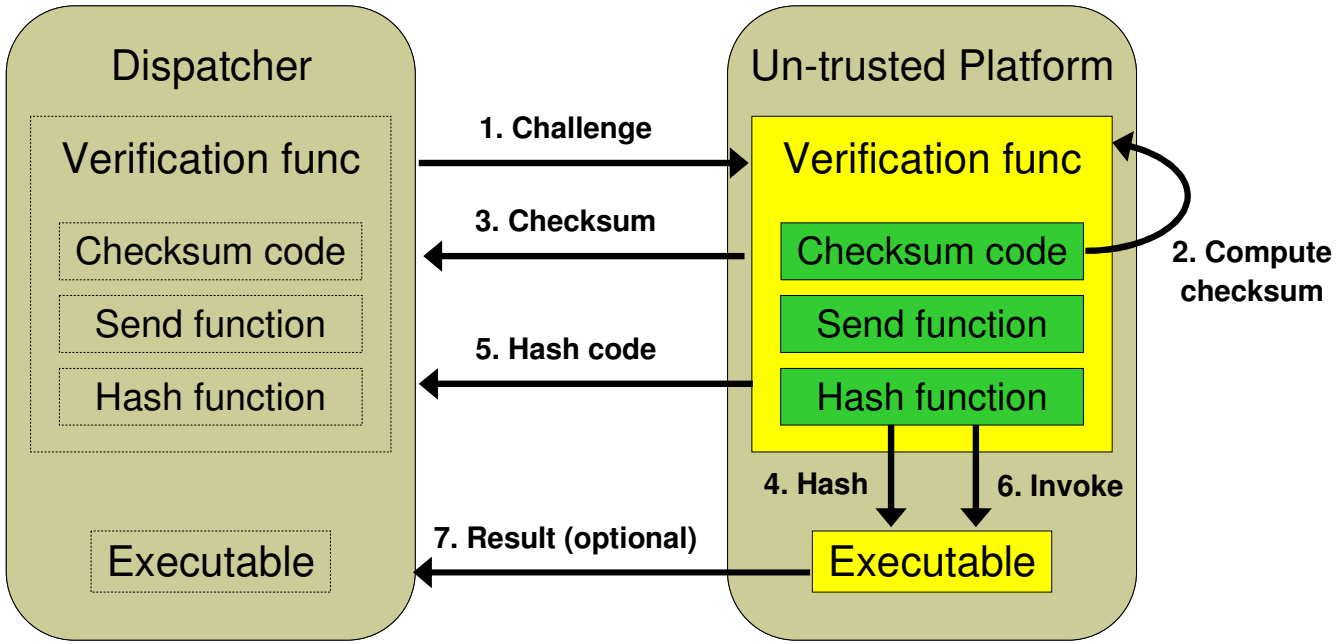
$$\neq (a_1 \oplus a_2) + (a_3 \oplus a_4)$$

Experimental results



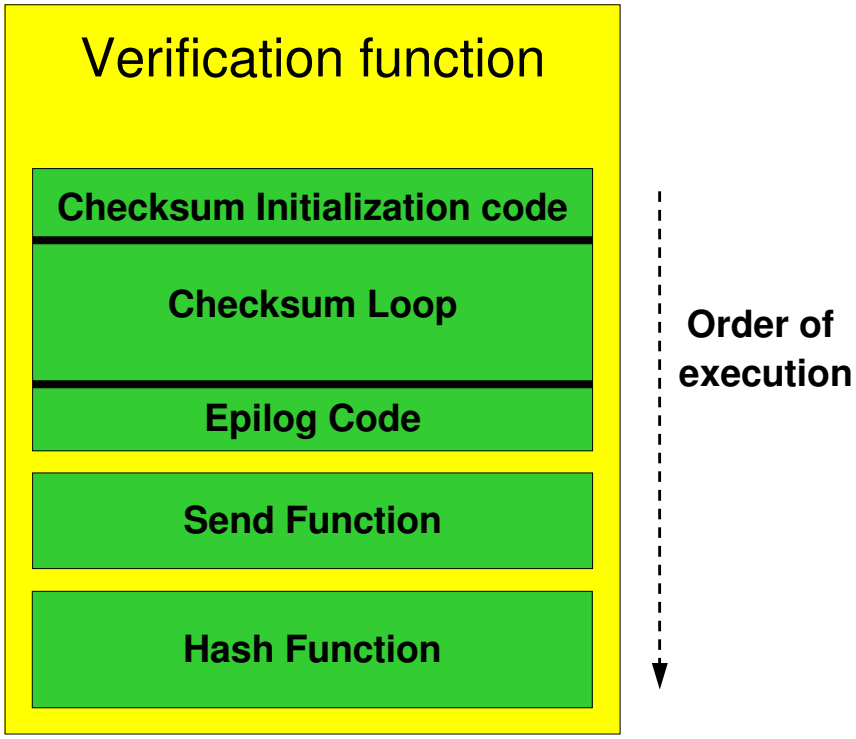
Pioneer: Dynamic root of trust

- It sets up the un-tampered execution environment.
- It computes a fingerprint of the whole verification function.
- It performs the integrity measurements on the executables.
- It execute the executable within the un-tampered execution environment.



Iterative checksum code

- Adversary who manipulates the input in every iteration of the checking function causes a constant time overhead per iteration.



Low variance in execution time

- Checking code is small enough to fit into L1 CPU instruction cache.
- Verification function is small enough to fit into L1 CPU data cache.
- Checksum code execute at the highest privilege level.
- All the maskable interrupts are turned off.
- Reduced number of non-issuable instruction (no out-of order execution in superscalar processors).
- No external function (os, library) is called (the OS is suspended).

Execution environment

- Turn off all the maskable interrupts
  - Success only if running at the highest privilege level.
  - Failure in case of lower privilege.
  - Time overhead if running in a software virtual machine monitor (*e.g.,* VMware).
- Register flags, program counter and data pointer are incorporated in each checksum iteration.
- Exception handler for all non-maskable interrupts is replaced with the “interrupt-return” instruction.
- Call stack is used to store part of the checksum during its computation.

How many iterations?

Adversary advantage **a**:

- Pre-load verification function into L1 CPU cache (no cache miss)
- Zero RTT

Adversary overhead **o**:

- Time required to forge registers and program/data pointer.
- Time to redirect memory accesses.

c: CPU speed.
n: number of iterations.

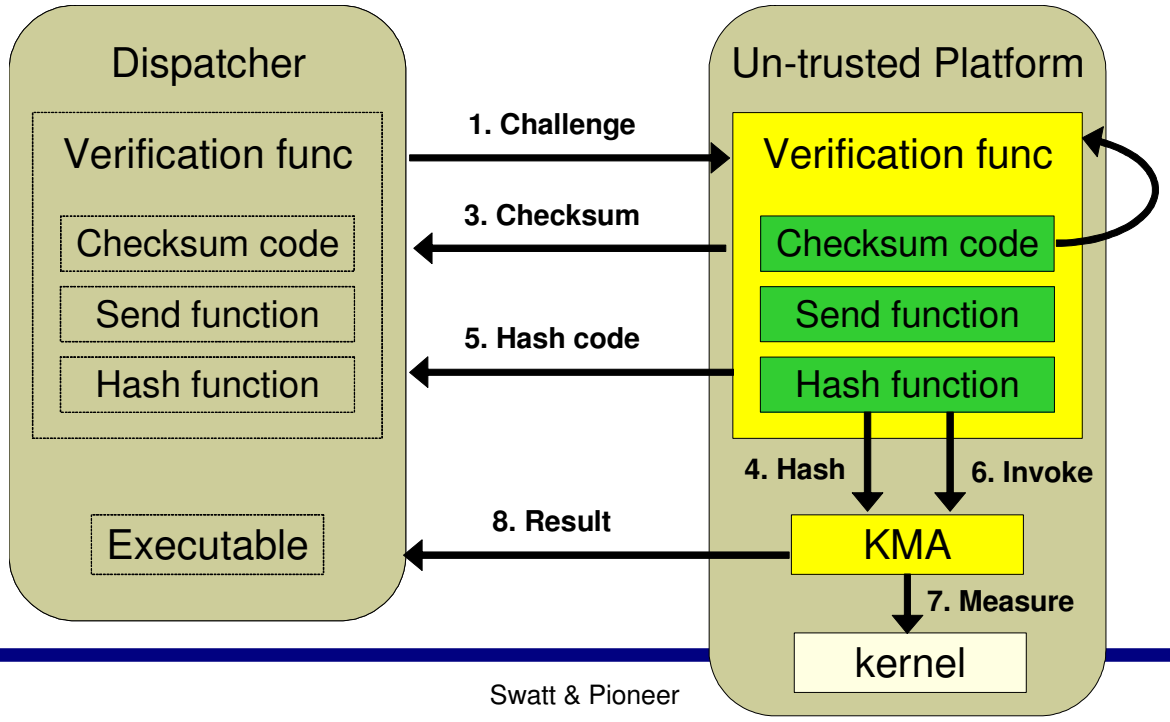
$$n > \frac{c * a}{o}$$

Experimental results

- RTT is evaluated considering the PING latency on different host in the LAN segment.
  - RTT < 0.25 ms
- Cache pre-warming time evaluated empirically
  - 0.0016 ms
- a = 0.2516 ms
- o = 0.6 CPU cycle per iteration
- n = 1,250,000 iterations (on 2.8Ghz CPU)
- To prevent false positives n is doubled (2,500,000 iterations).
- r = time to perform 2,500,000 iterations
- If dispatcher receive the answer after r + RTT it is considered in late.

Kernel rootkit detector

- Pioneer is used to guarantee the verifiable code execution of the Kernel Measurement Agent (KMA).
- KMA is used to compute the hash value of the running kernel.
- KMA runs at kernel privilege.
  - Kernel is hashed.
  - Module pointer is checked.
  - Kernel version is checked.
  - Return address is checked.



Experimental results

- Rootkit detector runs every 5 seconds.
- Computational and I/O intensive operations are used as benchmarks.
  - PostMark: file system benchmark.
  - Bunzip2: uncompress all the firefox source code.
  - Copy: copy of all the Linux source code (1.33 Gb).

Benchmark	Standalone	Rootkit detector	Overhead
PostMark	52	52.99	1.9%
Bunzip2	21.296	21.713	1.5%
Copy	373	385	3.2%

Open issues

- Formal proof of code optimality.
- Avoid that an adversary can use mathematical methods to generate a function that computes the same checksum when fed with the same input.
- Provide a checksum function which is CPU independent.
- Increase the time overhead for an attack.