Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems[1]

Mariano Ceccato

ceccato@itc.it

[1]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.
- TMP approach.
- Pioneer approach.
- Pioneer architecture.
- Adoption scenario: rootkit detector.

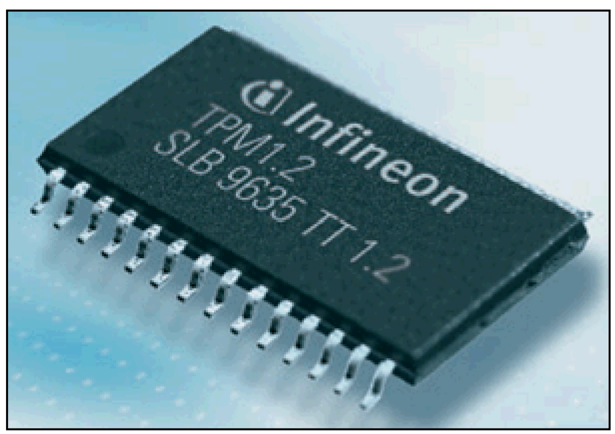
The problem

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.

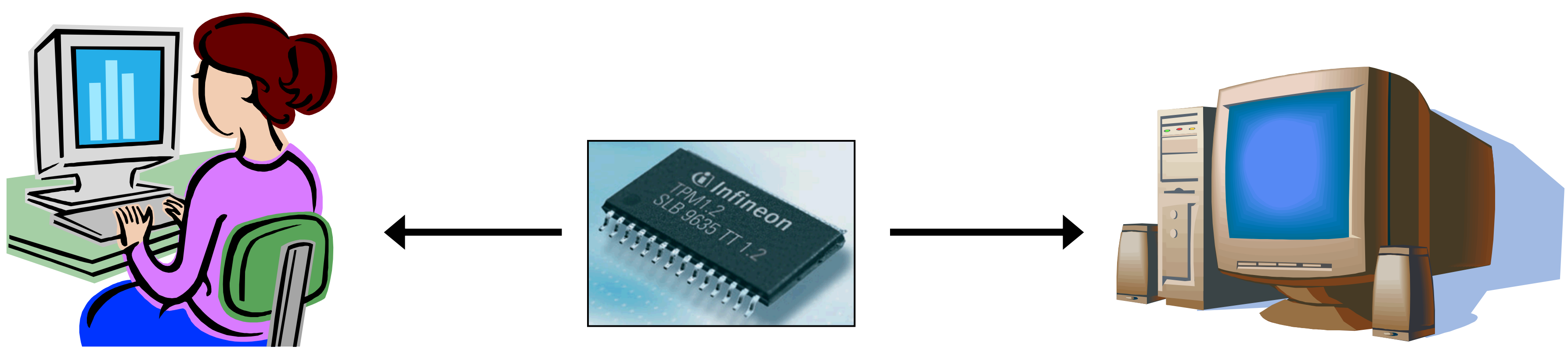
Trusted Platform Module

- TPM is a hardware security co-processor that provides some tamper resistant functions and secret keys.
- Secret keys generation.
- Cryptographic functions: encryption, decryption, hashing.
- Generation of ticks at a regular intervals (which can be signed by third party authorities)
- Monotonic counter function



Remote attestation

- TMP 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 TMP.
- Remote attestation allows a party to obtain assurance in the correct operation of a remote system.



Trusted Platform Module

- TMP based authentication can not be applied on legacy systems (where no special purpose hardware is available).
- Collision resistance property of SHA-1 hashing function has been compromised.
  - Tampered code with the same signature as the authentic one.
- When a fault is revealed it is not possible to fix it without replacing all the hardware.

Pioneer

- Software based primitive to verify code execution on an un-trusted legacy 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.
  - Communication link.

Assumptions

Dispatcher:

- It knows the exact hardware configuration of the un-trusted client.

Un-trusted client:

- Single CPU (not over-clocked).
- CPU does not support Symmetric Multi-Threading.

Communication channel:

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

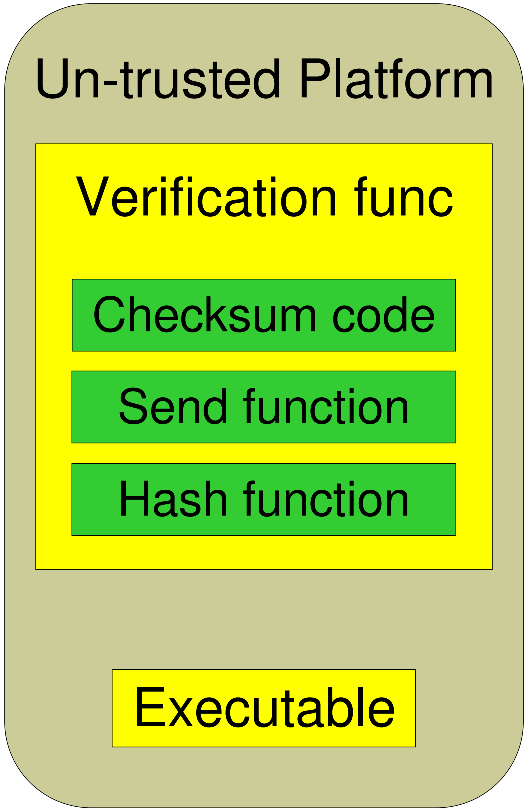
Attacker model

- The attacker has complete control of the software on the un-trusted platform (administrator privileges)
  - Applications.
  - Operative system.
- The attacker can not modify the hardware
  - He can not load malicious firmware on disk controllers or network interfaces.
  - He can not replace the CPU with a faster one.
  - He can not perform DMA-attacks.

Verification function

- The verification function checks itself.
- It performs the integrity measurements on the executables.

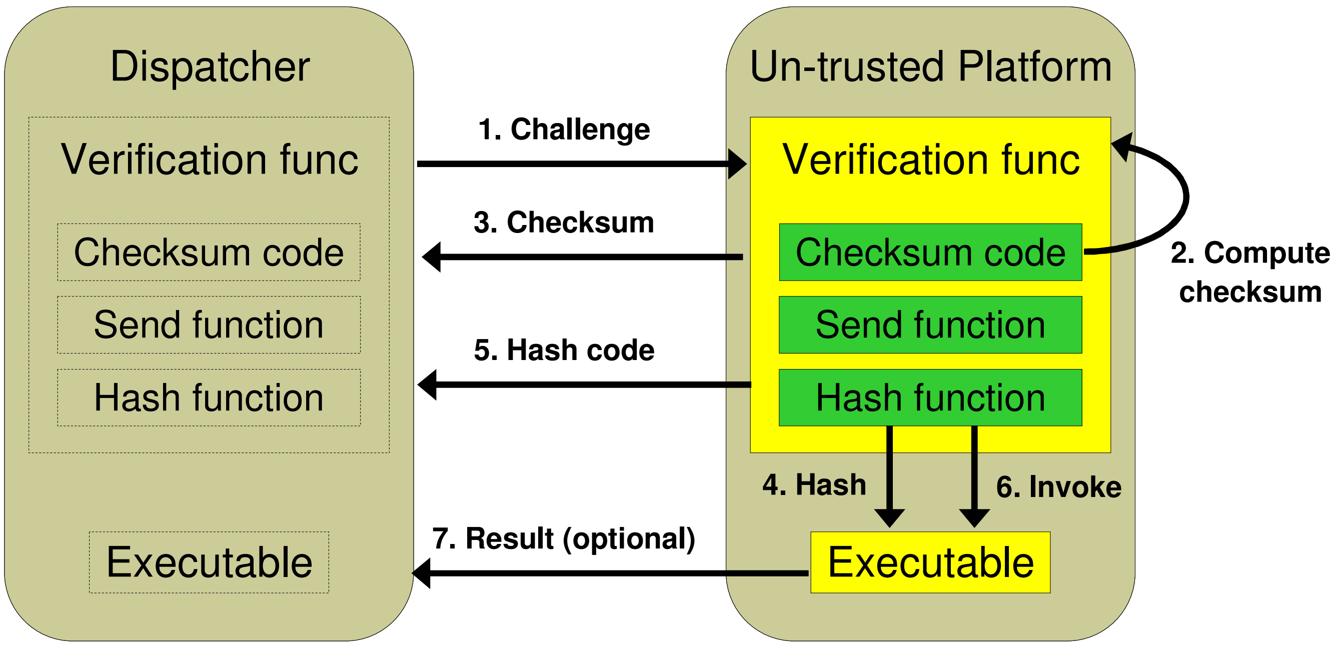
- Checksum code
  - It sets up the un-tampered environment.
  - It computes a fingerprint of the whole verification function.
  - Any attack will result in a noticeable time increase.
- Hash function
  - It depends on the challenge sent by the dispatcher.



The challenge-response protocol

- The dispatcher obtains the assurance that dynamic root of trust exists on the un-trusted platform.
- The dispatcher uses the dynamic root of trust to guarantee the verifiable code execution.

1.	$D:$	$t_1 \leftarrow$ current time, $nonce \xleftarrow{R} \{0,1\}^n$
	$D \rightarrow P:$	$\langle nonce \rangle$
2.	$P:$	$c \leftarrow \text{Checksum}(nonce, P)$
3.	$P \rightarrow D:$	$\langle c \rangle$
	$D:$	$t_2 \leftarrow$ current time if $(t_2 - t_1 > \Delta t)$ then exit with failure else verify checksum c
4.	$P:$	$h \leftarrow \text{Hash}(nonce, E)$
5.	$P \rightarrow D:$	$\langle h \rangle$
	$D:$	verify measurement result h
6.	$P:$	transfer control to E
7.	$E \rightarrow D:$	$\langle$result (optional)$\rangle$



Time-optimal implementation of 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.
  - Multiple instructions are issued in a superscalar processor.
  - No other issue slot are available for malicious code.

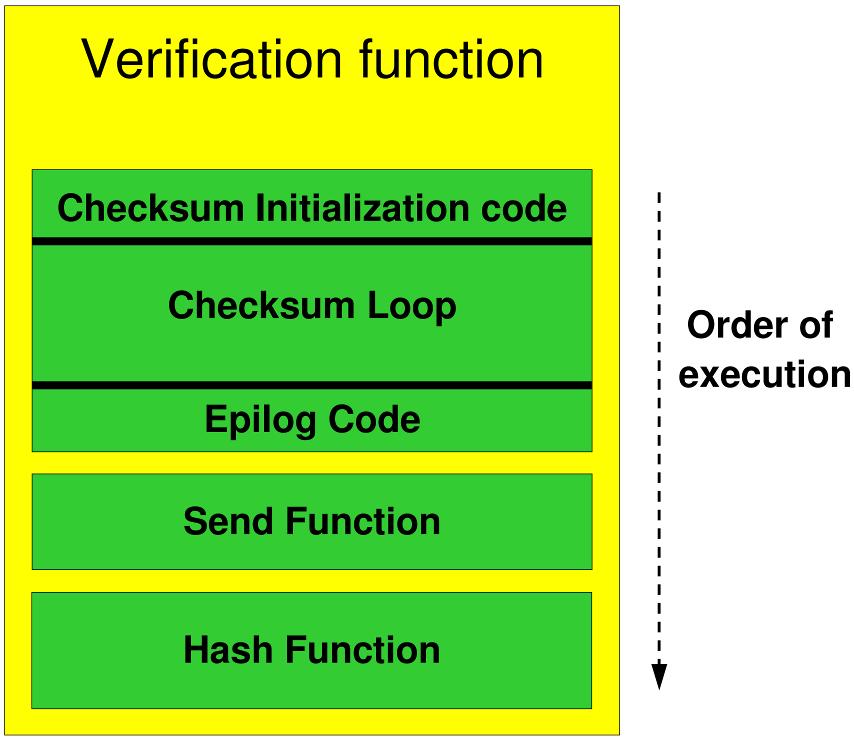
a_1
a_2
a_3
a_4
…

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

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

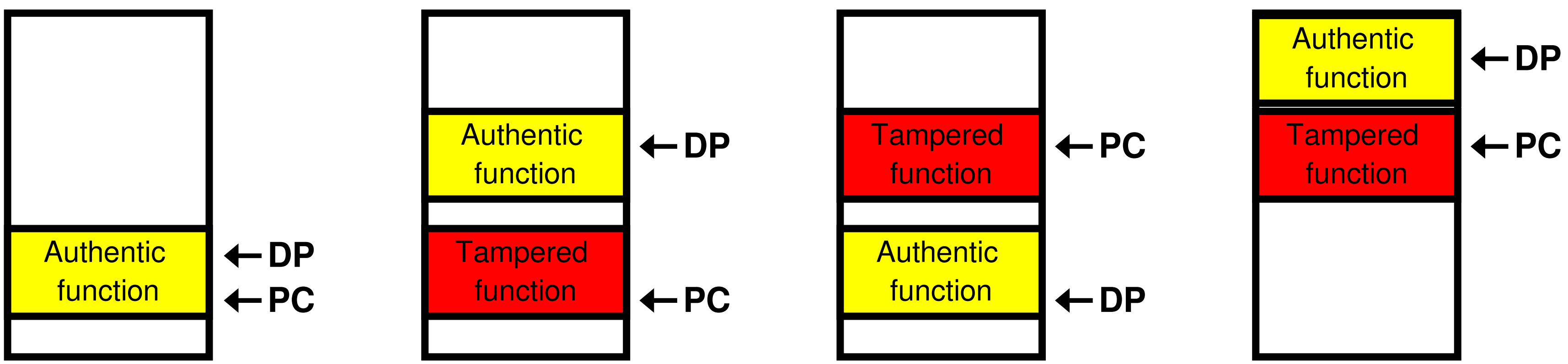
Iterative checksum code

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



Checksum includes CPU state

- The adversary compute the checksum on a correct copy of the tampered verification function (Memory copy attack).
- Incorporate both PC and DP into the checksum computation, so when they are required the adversary loses time to forge them.

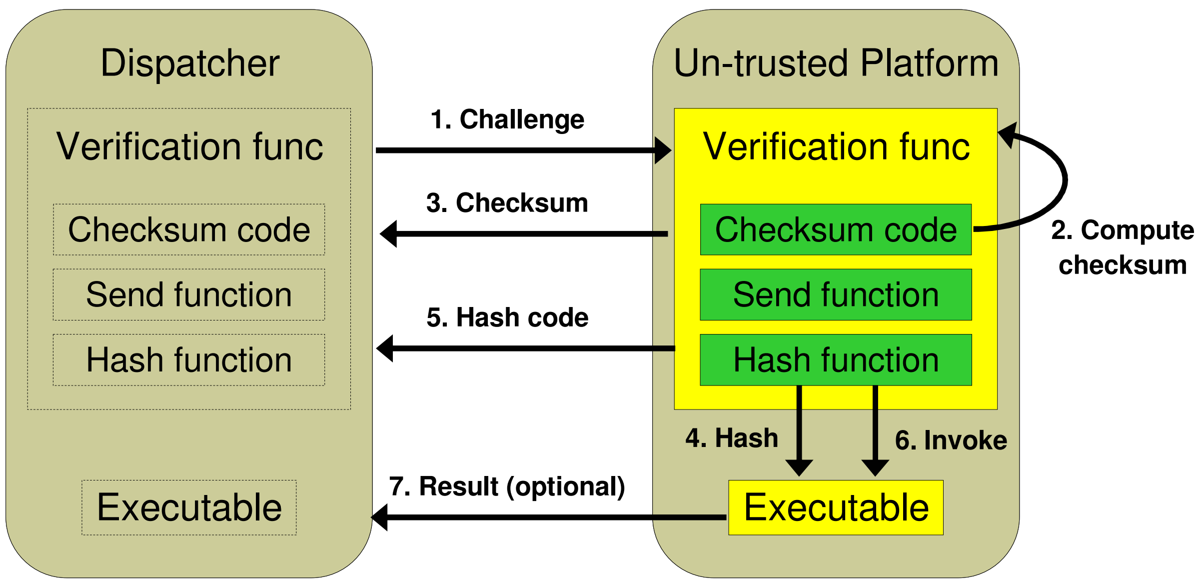


Low variance in execution time

- 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.

Dispatcher challenge

- The checksum depends on the challenge sent by the dispatcher.
  - The adversary can not pre-compute the checksum.
  - Challenge is used to initialize a pseudo-random number generator used in pseudo-random memory traversal.
  - Challenge is the initialization value for the checksum.



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 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 can pre-load verification function into L1 CPU cache (no cache miss) and have a zero RTT
  - Adversary time advantage (a).
- Adversary overhead per iteration (o) .
  - Total overhead increases linearly with the number of iterations (n*o/c).
- CPU clock speed (c).

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

Experimental results

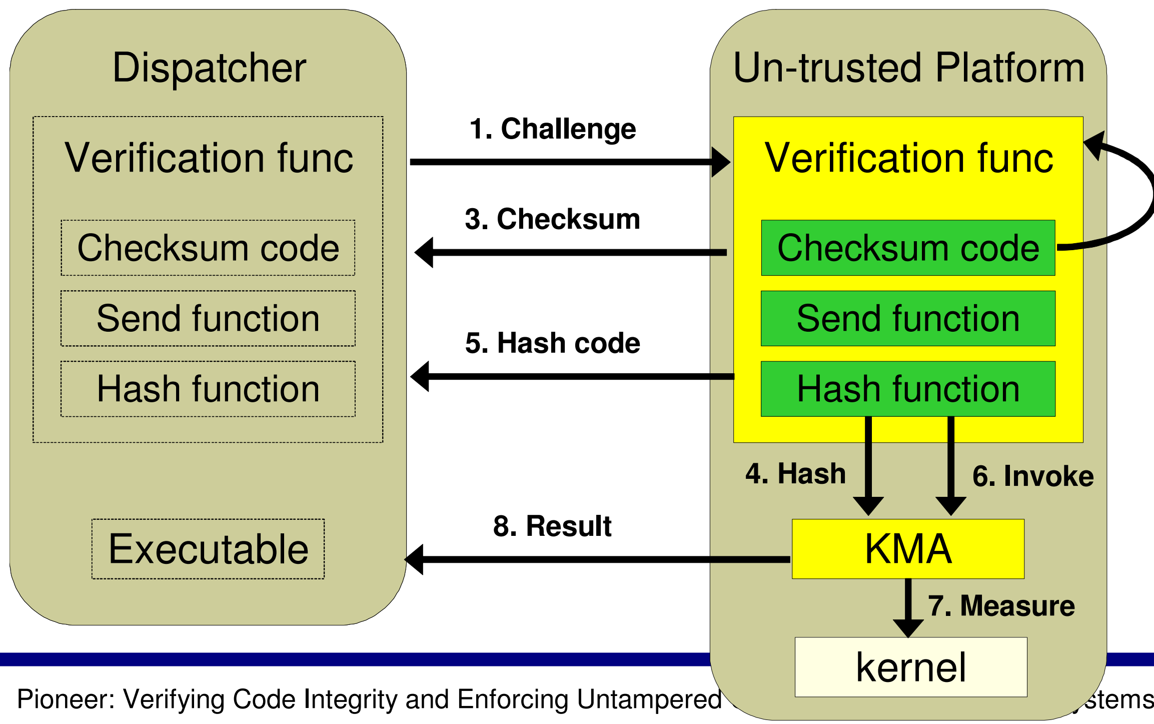
- RTT is evaluated considering the PING latency on different host in the LAN segment.
  - RTT < 0.25 ms
- Cache pre-warming time evaluated empirically
  - 0.0016 ms
- a = 0.2516 ms
- o = 0.6 CPU cycle per iteration
- n = 1,250,000 iterations (on 2.8Ghz CPU)
- To prevent false positives n is doubled (2,500,000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

Rootkits

- Rootkit is a software installed by an intruder on a host that allows the intruder to gain privileged access to the host, while remaining undetected.
  - Some rootkits do not modify the kernel (easy to locate).
  - Some rootkits do modify the kernel (kernel can not be trusted to locate them).

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).

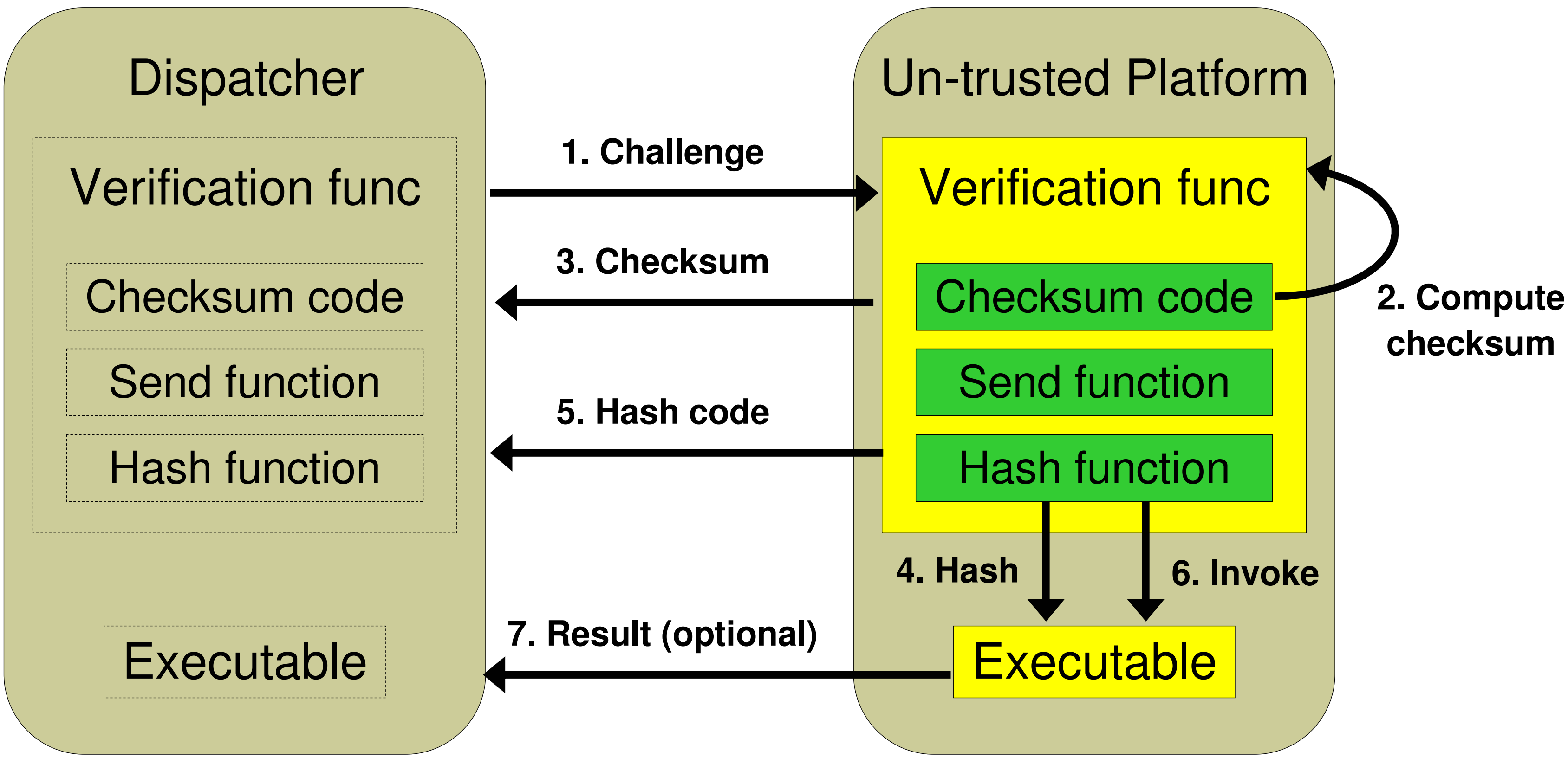
Benchmark	Standalone	Rootkit detector	Overhead
PostMark	52	52.99	1.9%
Bunzip2	21.296	21.713	1.5%
Copy	373	385	3.2%

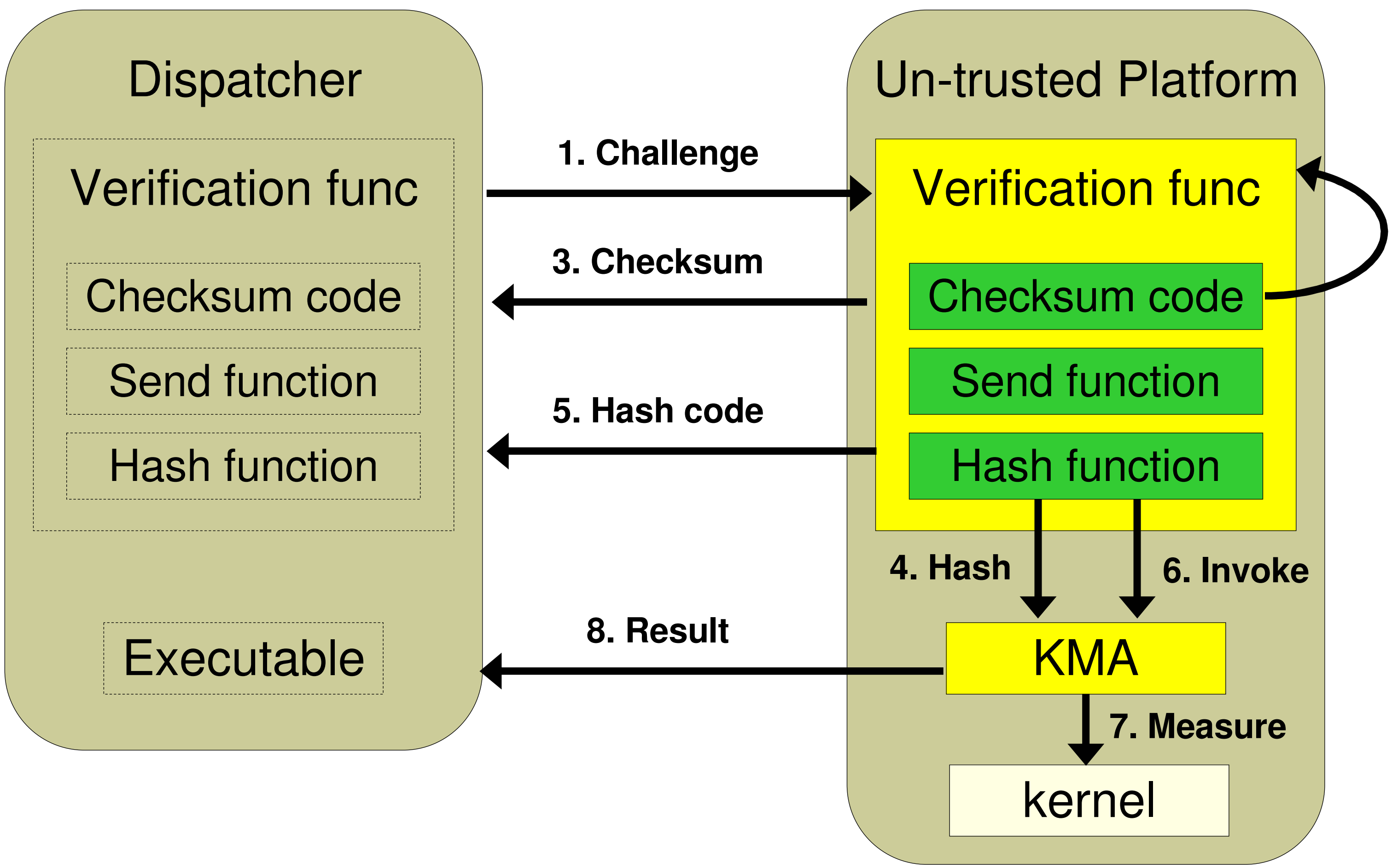
Open issues

- 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.

End of slide show, click to exit.

Pioneer architecture





Dispatcher
Verification func
Checksum code
Send function
Hash function
Executable
1. Challenge
3. Checksum
5. Hash code
8. Result
Un-trusted Platform
Verification func
Checksum code
Send function
Hash function
4. Hash
6. Invoke
KMA
7. Measure
kernel