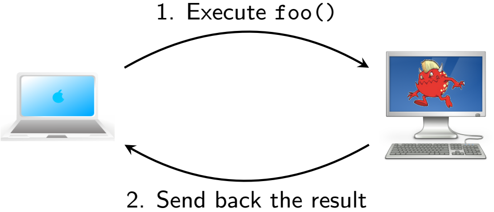
Conqueror: tamper-proof code execution on legacy systems

Lorenzo Martignoni[1] **Roberto Paleari**[2] Danilo Bruschi[2]

[1]Università degli Studi di Udine [2]Università degli Studi di Milano

7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '10)

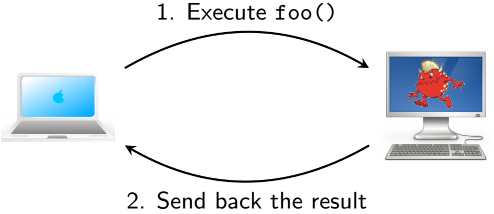
Verify the integrity of a piece of code executing in an untrusted system



1. Execute `foo()`

2. Send back the result

Verify the integrity of a piece of code executing in an untrusted system

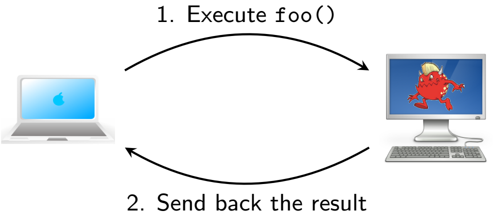


1. Execute `foo()`

2. Send back the result

1. `foo()` has been executed?
2. Is the result of `foo()` authentic?

Verify the integrity of a piece of code executing in an untrusted system



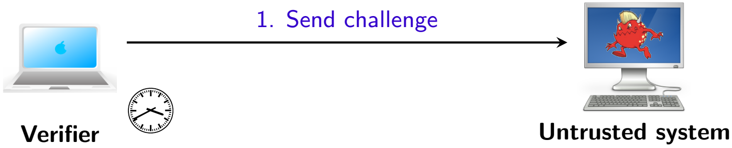
1. Execute `foo()`

2. Send back the result

1. `foo()` has been executed?
2. Is the result of `foo()` authentic?

Can we prove 1 and 2 with a **pure software-based** solution?

Software-based attestation through challenge-response



1. Send challenge

Verifier

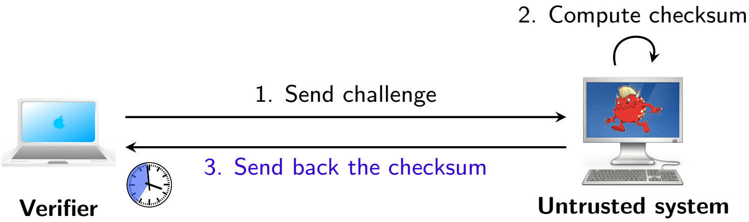
Untrusted system

- The verifier challenges the untrusted system (to compute a checksum)

Software-based attestation through challenge-response

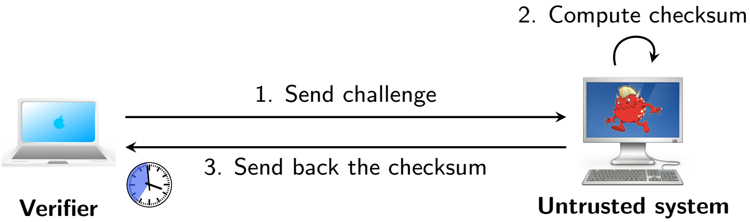
- The untrusted system executes the *checksum function*
- Should be executed at the **highest level of privilege**
- Should execute **without any interruption**

Software-based attestation through challenge-response



- The checksum must be received within a **time interval**
- Time is measured by an external entity (the *verifier*)
- If the checksum is wrong or the timeout has expired, attestation fails

Software-based attestation through challenge-response



2. Compute checksum

1. Send challenge

3. Send back the checksum

Verifier

Untrusted system

- The checksum must be received within a **time interval**
- Time is measured by an external entity (the *verifier*)
- If the checksum is wrong or the timeout has expired, attestation fails

Any attempt to tamper the execution environment results in a noticeable overhead in checksum computation

Pioneer: State-of-the-art software-based attestation solution

Characteristics

- Applies to legacy systems (e.g., no TPM)
- Checksum function is known *a priori*
- Implementation of the checksum function is *time-optimal*
- The *challenge* is in a seed to initialize the checksum function

Limitations

- Researchers found ways to thwart Pioneer (e.g., through TLBs desynchronization)
- Does not take into account hypervisor-based attackers

Pioneer: Verifying Code Integrity and Enforcing Un-tampered Code Execution on Legacy Systems
(Sheshadri, Pradeep, Mark Luck, Doorn, Perrig, Elaine)

Conqueror: Bullet-proof software-based code attestation

Features

- Legacy systems (e.g., no TPM)
- Immune to **all** the attacks that are known to defeat Pioneer
- Effective even against **hypervisor-based attackers**

Conqueror: Bullet-proof software-based code attestation

Features

- Legacy systems (e.g., no TPM)
- Immune to **all** the attacks that are known to defeat Pioneer
- Effective even against **hypervisor-based attackers**

Threat model

- Attacker cannot operate in SMM
- No hardware-based attacks (e.g., DMA attacks)
- Single thread of execution (e.g., no SMP)
- Attacker cannot leverage a pristine or more powerful system

How Conqueror works?

- Variation of the traditional challenge-response scheme
- The challenge is not a seed, but consists in the **whole** checksum function
- The checksum function is:
  1. Generated on demand
  2. Obfuscated
  3. Self-decrypting

Rationale behind Conqueror

- Conqueror's checksum functions are **not** optimal
- As functions are generated on demand and obfuscated, attackers must first analyze them

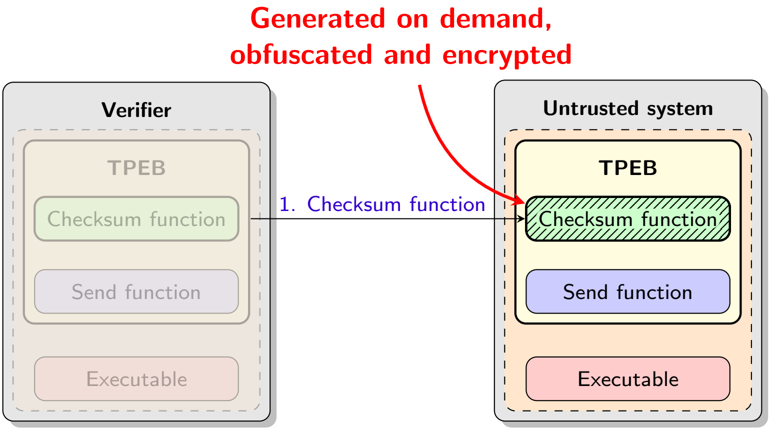
Our claim

An attacker has two options:

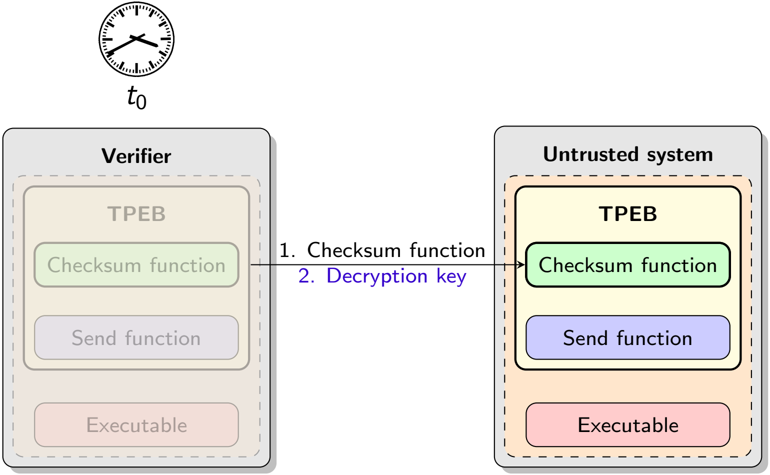
- Static analysis
- Dynamic analysis

Both static and dynamic attacks introduce a noticeable overhead in checksum computation

Conqueror protocol



Conqueror protocol



t_0

Verifier

TPEB

Checksum function

Send function

Executable

1. Checksum function

2. Decryption key

Untrusted system

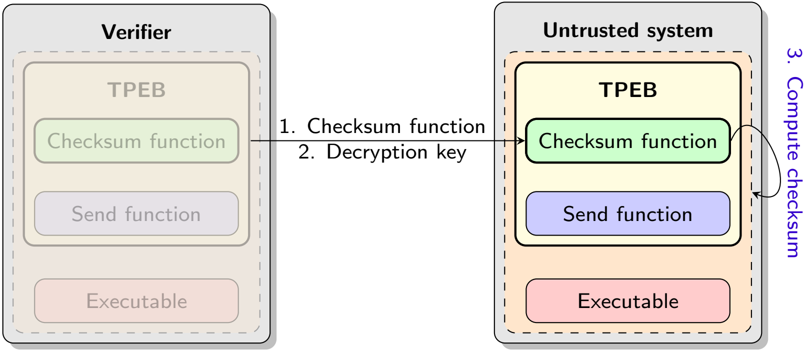
TPEB

Checksum function

Send function

Executable

Conqueror protocol



Verifier

TPEB

Checksum function

Send function

Executable

1. Checksum function

2. Decryption key

Untrusted system

TPEB

Checksum function

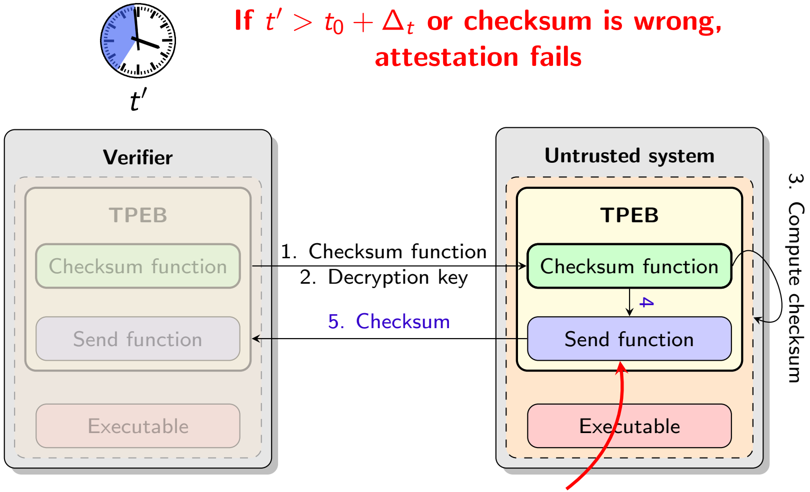
Send function

Executable

3. Compute checksum

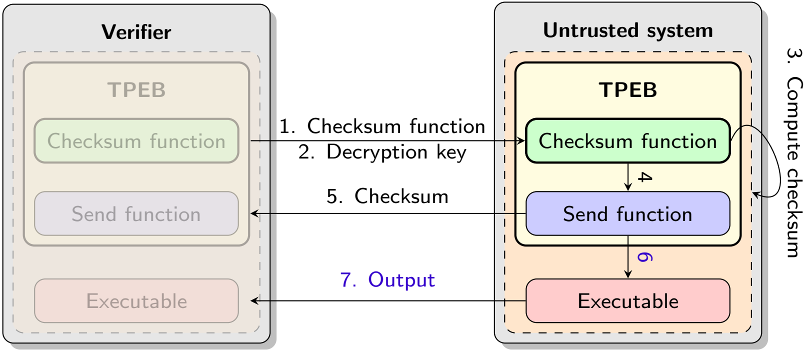
Conqueror protocol

If $t' > t_0 + \Delta_t$ or checksum is wrong, attestation fails

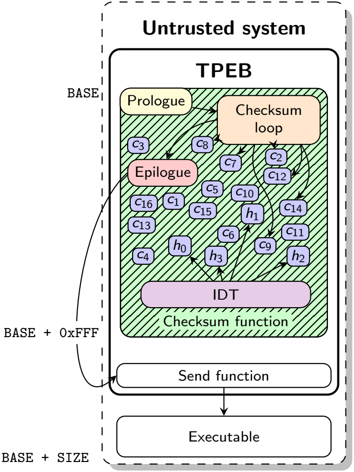


Hardware-dependent

Conqueror protocol

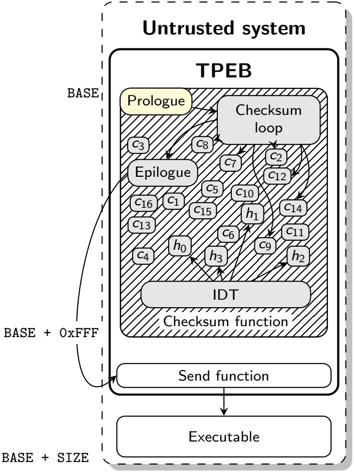


Tamper-**P**roof **E**nvironment **B**ootstrapper (TPEB)



- Attestation of the memory region $[\mathtt{BASE}, \mathtt{BASE} + \mathtt{SIZE})$
- Attestation of the environment:
  - Maximum privilege
  - Interrupts disabled
  - No hypervisor

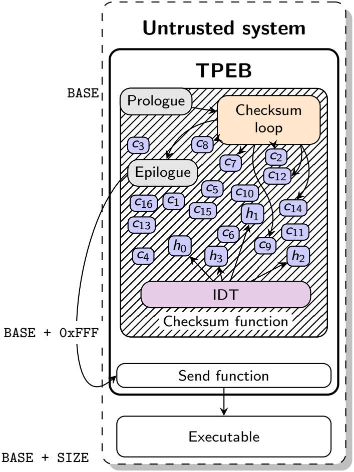
Tamper-**P**roof **E**nvironment **B**ootstrapper (TPEB)



Prologue

1. Disables maskable interrupts
2. Decrypts the rest of the page
3. Installs custom interrupt handlers

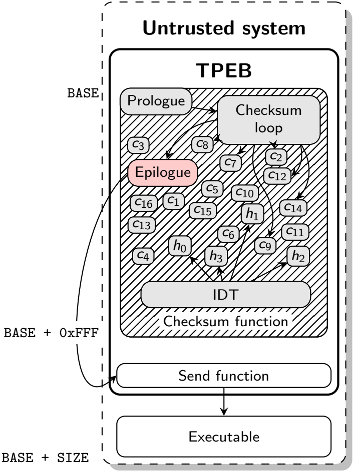
Tamper-**P**roof **E**nvironment **B**ootstrapper (TPEB)



Checksum loop

- Made of different *gadgets*
- They update the running value of the checksum according to the content of a memory location
- Gadgets are selected and combined randomly
- Gadgets are obfuscated

Tamper-**P**roof **E**nvironment **B**ootstrapper (TPEB)



Epilogue

- Invokes the send function
- Transfers the control to the executable

Checksum loop

- Iterate the memory to attest in a pseudorandom fashion

```
for (i = 0, j = 0; i < ITERATIONS; i++) {
   x = seed(i) % (SIZE / 4);
   do {
      x = (x + (x*x | 5)) % (SIZE / 4);
      checksum_gadget[j++ % GADGETS](BASE + x*4);
   } while (x != seed(i) % (SIZE / 4));
}
```

Checksum loop

- Iterate the memory to attest in a pseudorandom fashion
- The content of each location is fed to a different gadget, that updates the checksum

```
for (i = 0, j = 0; i < ITERATIONS; i++) {
   x = seed(i) % (SIZE / 4);
   do {
      x = (x + (x*x | 5)) % (SIZE / 4);
      checksum_gadget[j++ % GADGETS](BASE + x*4);
   } while (x != seed(i) % (SIZE / 4));
}
```

Checksum loop

- Iterate the memory to attest in a pseudorandom fashion
- The content of each location is fed to a different gadget, that updates the checksum
- The whole memory traversal process is repeated multiple times

```
for (i = 0, j = 0; i < ITERATIONS; i++) {
   x = seed(i) % (SIZE / 4);
   do {
      x = (x + (x*x | 5)) % (SIZE / 4);
      checksum_gadget[j++ % GADGETS](BASE + x*4);
   } while (x != seed(i) % (SIZE / 4));
}
```

Gadgets

Active gadgets

- Intentionally executed by the checksum function
- Update the checksum
- Verify the trustworthiness of the environment

Passive gadgets

- Executed on interrupts and exceptions
- Corrupt the checksum when unexpected events occur
- Registered by installing a custom interrupt descriptor table

Active gadgets: Plain checksum computation

- Most frequently used gadget
- Simply updates the checksum

```
mov ADDR, %eax
mov (%eax), %eax
xor $0xa23bd430, %eax
add %eax, CHKSUM+4
```

Active gadgets: IDT attestation

- IDT is part of the TPEB
- Normal checksum computation attests the *content* of the IDT
- Need a gadget to attest the *address* of the IDT

```
mov ADDR, %eax
mov (%eax), %eax
add %eax, CHKSUM+8
sidt IDTR
mov IDTR+2, %eax
xor $0x6127f1, %eax
add %eax, CHKSUM+8
```

Active gadgets: System mode attestation

- Prevent the computation of the checksum from user mode
- Update the checksum through privileged instructions
- If executed in user mode, these instructions raise an exception

```
mov ADDR, %eax
mov (%eax), %eax
xor $0x1231d22, %eax
mov %eax, %dr3
mov %dr3, %ebx
add %ebx, CHKSUM
```

Active gadgets: Instruction and data pointers attestation

- Based on *self-modifying code*
- Prevent *memory copy attacks* (e.g., TLB desynchronization)
- Attest that the VA ↔ PHY holds for read, write and fetch operations

```
  mov ADDR, %eax
  mov (%eax), %eax
  lea l_smc, %ebx
  roll $0x2, 0x1(%ebx)
l_smc:
  xor $0xdeadbeef, %eax
  add %eax, CHKSUM+4
```

Active gadgets: Hypervisor detection

- Rich ongoing debate on this topic . . .
- Exploit timing attacks to detect running HVMs
- Execute instruction that *unconditionally* trap to the hypervisor

```
mov ADDR, %eax
mov (%eax), %ebx
vmlaunch
xor $0x7b2a63ef, %ebx
sub %ebx, CHKSUM+8
```

Evaluation

Experimental setup

- Prototype for Microsoft Windows XP (32-bit)
  - **Verifier**: user/kernel component
  - **Untrusted system**: device driver
- Two scenarios:
  1. *Static* attack (e.g., reverse engineering of the checksum function)
  2. *Dynamic* hypervisor-based attack (most powerful attacker)

Parameters

- $\sim$ 100 gadgets, minimum 5% for hypervisor detection
- Rely on a **trusted system** to estimate network RTT and maximum checksum computation time
- Trusted and untrusted systems have the same hardware configuration

Estimating the maximum checksum computation time

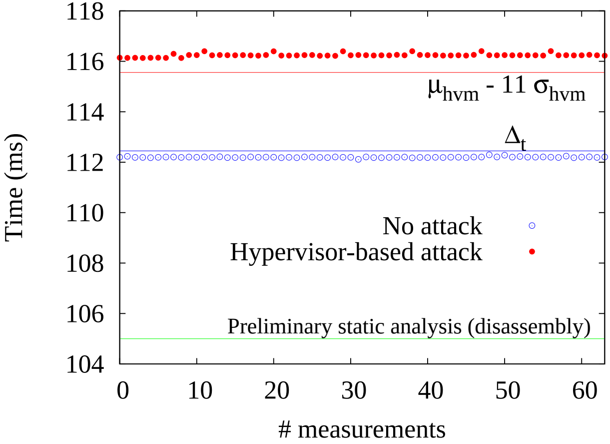
- Execution time of checksum functions can be precomputed using a trusted system
- Use Chebyshev's inequality to estimate an upper bound on computation

$$Pr(\mu - \sigma \leq \boxed{X} \leq \mu + \sigma) \geq 1 - \frac{1}{\lambda^2}$$

Computation time (including RTT)

- Upper bound is $\Delta_t = \mu + \lambda\sigma$
- We choose $\lambda = 11$, to obtain a confidence $> 99\%$
- For a given checksum function, we estimate Δ_t by challenging the trusted system multiple times

Checksum computation time



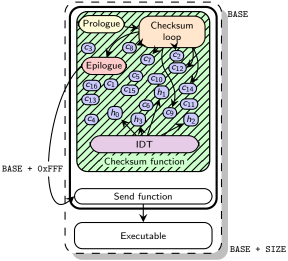
- No checksum was forged in time to be considered valid
- No authentic checksum was considered forged

Conclusions

- We extended current state-of-the-art code attestation solutions
- Prototype implementation of our attestation scheme
- Conqueror is the basic building block of our next projects
  - "*Dynamic and Transparent Analysis of Commodity Production Systems*" (ASE 2010)
  - "*Live and Trustworthy Forensic Analysis of Commodity Production Systems*" (RAID 2010)

Conqueror

Tamper-proof code execution on legacy systems



Thank you!
Any questions?

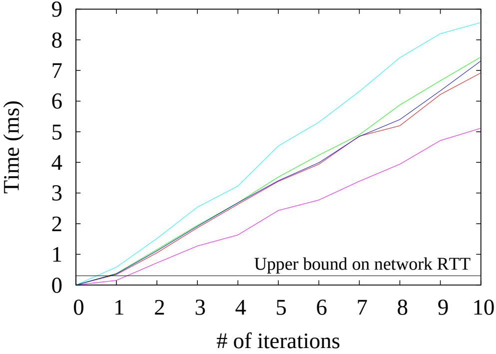
Roberto Paleari

`roberto@security.dico.unimi.it`

Backup slides

Estimating the ideal number of memory iterations

- Time overhead suffered by a hypervisor-based attacker, using 5 checksum functions
- We assume the attacker has RTT = 0



- Two iterations of the checksum loop are enough
- To prevent false negatives, we perform **four iterations**