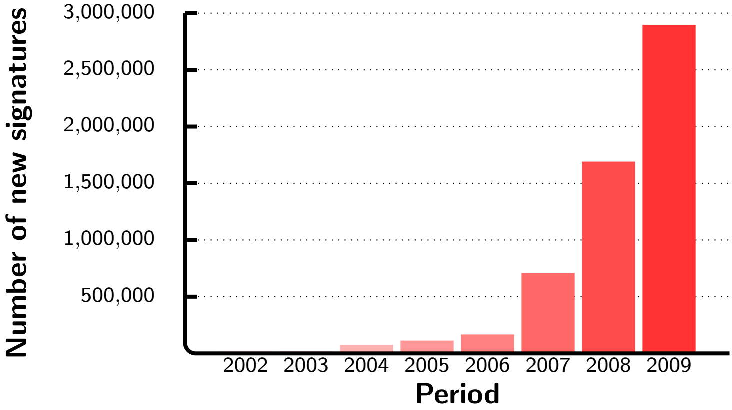
Dealing with next-generation malware

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano

Doctor of Philosophy in Computer Science

Advisor: Prof. D. Bruschi
PhD Candidate: Roberto Paleari

The rise of malicious code



The rise of malicious code

Today malware is a **very lucrative** activity

The rise of malicious code

Who lasts longer earns the most . . .

Long lasting malware

- Spread/replicate fast
- Hide the presence on the system
- Obfuscate the code (e.g., encryption, polymorphism, metamorphism)

Long lasting malware

- Spread/replicate fast
- Hide the presence on the system
- Obfuscate the code (e.g., encryption, polymorphism, metamorphism)

Traditional signature-based approaches are **not effective anymore**!

Cyveillance testing finds AV vendors detect on average less than 19% of malware attacks

Further testing reveals that even after 30 days, detection rates averaged only 61.7%

ARLINGTON, Va., August 4, 2010 -- Cyveillance, a world leader in cyber intelligence, today announced the availability of their most recent Internet security report, "Malware Detection Rates for Leading AV Solutions: A Cyveillance Analysis." The report reveals that traditional antivirus (AV) vendors continue to lag behind online criminals when it comes to detecting and protecting against new and quickly evolving threats on the Internet. Cyveillance testing[1] shows that even the most popular AV signature-based solutions detect on average less than 19% of malware threats. That detection rate increases only to 61.7% after 30 days.

Current trend for malware analysis and detection

Static analysis is either too onerous or impossible

(malware is obfuscated & self-modifying)

Current trend for malware analysis and detection

~~Static analysis is either too onerous or impossible~~
(malware is obfuscated & self-modifying)

Dynamic, behavior-based malware analysis

Limitations of dynamic approaches

Incompleteness

- The analysis involves a limited number of program paths
-  may behave maliciously only in very specific circumstances

Limitations of dynamic approaches

Incompleteness

- The analysis involves a limited number of program paths
-  may behave maliciously only in very specific circumstances

Non-transparency

- The analysis tool can be detected
- If  detects the analyzer, it behaves like 

Limitations of dynamic approaches

How to perform post-infection analysis?

- If the host has already been compromised,  could tamper with the execution of the analysis tool

Limitations of dynamic approaches

How to perform post-infection analysis?

- If the host has already been compromised,  could tamper with the execution of the analysis tool

High run-time overhead

- End hosts have strict real-time constraints
- If the analysis takes too much, the detector assumes a suspicious program is 

Research goal

Next-generation malware is a new category of highly-sophisticated malicious threats

↓

Limitations of anti-malware tools are exacerbated when dealing with next-generation malware

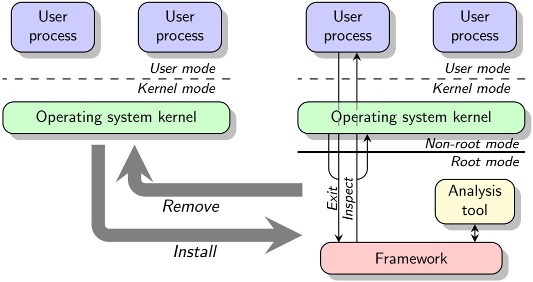
Goal

To propose malware analysis & detection infrastructures that overcome the limitations of current technology

Contributions at a glance

1. Dynamic and Transparent Analysis of Commodity Production Systems
   (ASE 2010)
2. Conqueror: Tamper-proof Code Execution on Legacy Systems
   (DIMVA 2010)
3. Live and Trustworthy Forensic Analysis of Commodity Production Systems
   (RAID 2010)
4. A Framework for Behavior-based Malware Analysis in the Cloud
   (ICISS 2009)

Transparent and efficient analysis



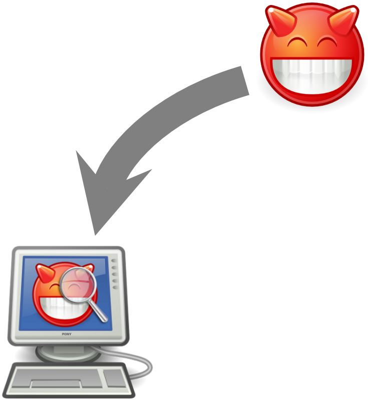
Dynamic and Transparent Analysis of Commodity Production Systems (ASE 2010)

Problem definition

How to monitor the execution of a suspicious program?
(worst-case scenario: kernel-level malware)

Problem definition

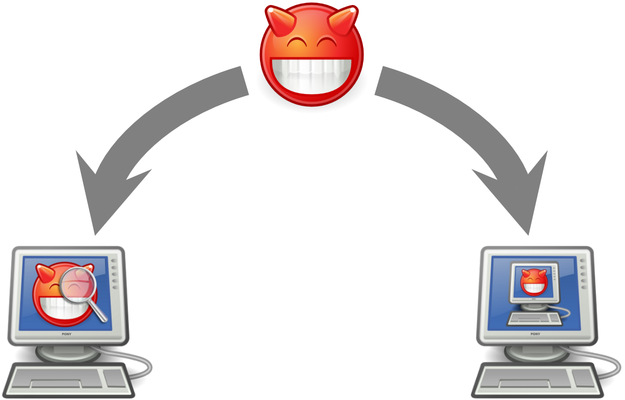
How to monitor the execution of a suspicious program?
(worst-case scenario: kernel-level malware)



Kernel-based analysis

Problem definition

How to monitor the execution of a suspicious program?
(worst-case scenario: kernel-level malware)



Kernel-based analysis | **Out-of-the-box analysis**

Kernel-based approaches

- The analysis tool is implemented as a **kernel module**
- To analyze kernel-level code, these approaches leverage another kernel-level module . . .

Kernel-based approaches

- The analysis tool is implemented as a **kernel module**
- To analyze kernel-level code, these approaches leverage another kernel-level module . . .

. . . it is like a dog chasing its tail!

Out-of-the-box approaches

- The analyzer leverages **VM-introspection techniques**
- The target system must be **already running inside a VM**

Out-of-the-box approaches

- The analyzer leverages **VM-introspection techniques**
- The target system must be **already running inside a VM**

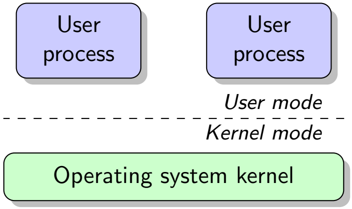
 **is often able to detect VMs!**

How to automatically generate procedures to detect CPU emulators (WOOT 2009)

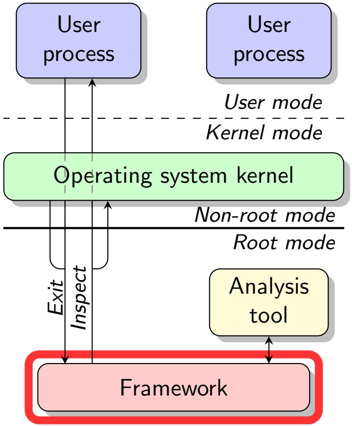
Our solution

Exploit hardware support for virtualization to achieve both **efficiency** and **transparency**

Overview of the framework

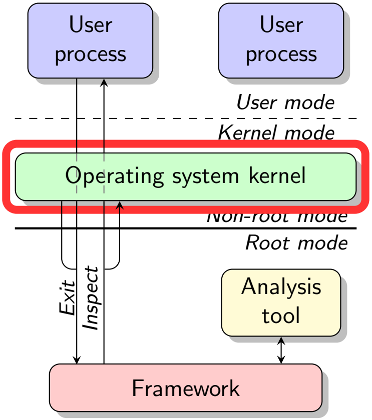


Overview of the framework



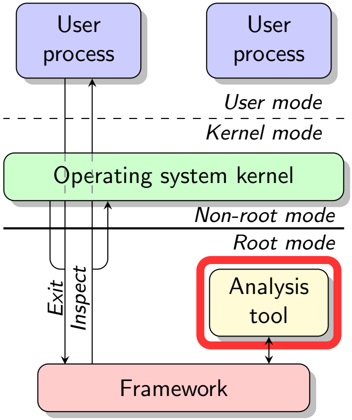
The framework is installed **as the target system runs**. It is completely separated and more privileged than the analyzed OS

Overview of the framework



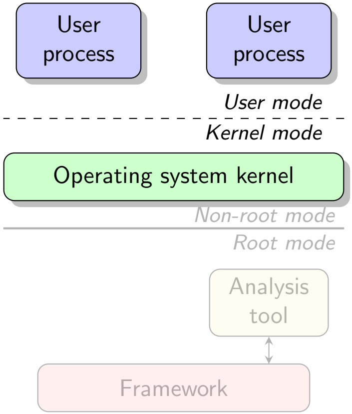
The analyzed OS **needs not to be modified** at all
(i.e., the approach can be applied to closed-source OSes)

Overview of the framework



The analysis tool runs in an **isolated execution environment**
(a defect in the tool does not affect the stability of the OS)

Overview of the framework



At the end of the analysis, the infrastructure can be **removed on-the-fly**

An application: HyperDbg

- A **transparent kernel debugger** built on top of our framework
- Offers standard debugging features, at the kernel-level
  (e.g., breakpoints, watchpoints, single-stepping)

An application: HyperDbg

- A **transparent kernel debugger** built on top of our framework
- Offers standard debugging features, at the kernel-level
  (e.g., breakpoints, watchpoints, single-stepping)

What are the key advantages of HyperDbg?

vs

Even kernel-level malware cannot affect its execution

An application: HyperDbg

- A **transparent kernel debugger** built on top of our framework
- Offers standard debugging features, at the kernel-level
  (e.g., breakpoints, watchpoints, single-stepping)

What are the key advantages of HyperDbg?

vs

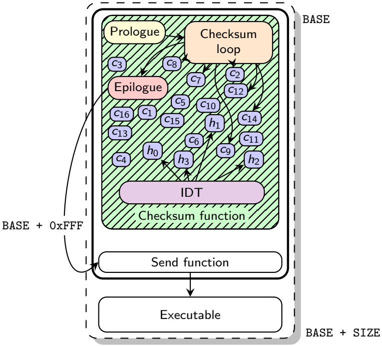
The target needs not to be running inside a VM

An application: HyperDbg

- A **transparent kernel debugger** built on top of our framework
- Offers standard debugging features, at the kernel-level
  (e.g., breakpoints, watchpoints, single-stepping)

http://code.google.com/p/hyperdbg/

Software-based code attestation



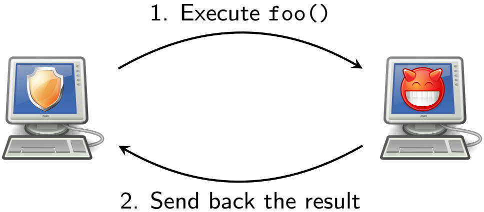
Conqueror: Tamper-proof Code Execution on Legacy Systems (DIMVA 2010)

Problem definition

How to guarantee that the execution of an anti-malware tool has not been tampered?

Problem definition

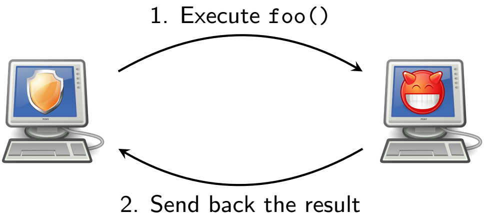
How to guarantee that the execution of an anti-malware tool has not been tampered?



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

Problem definition

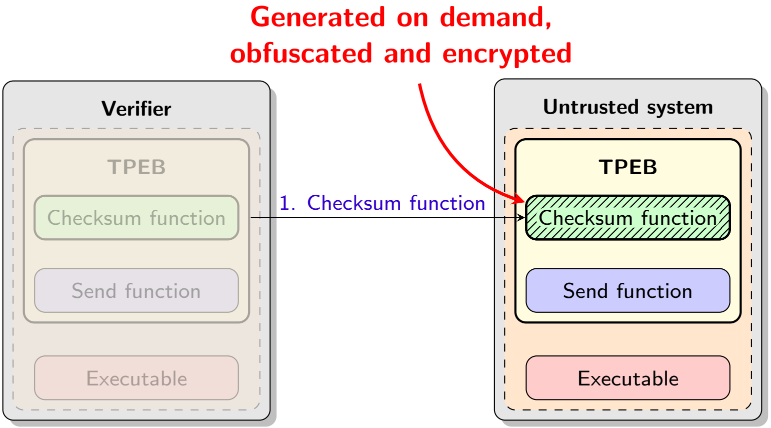
How to guarantee that the execution of an anti-malware tool has not been tampered?



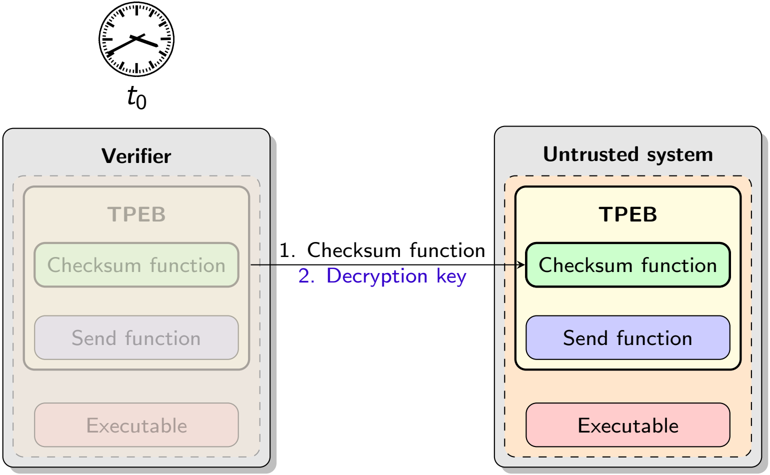
1. `foo()` has been executed?
2. Is the result of `foo()` authentic?

Can we prove (1) + (2) with a **pure software-based** solution?

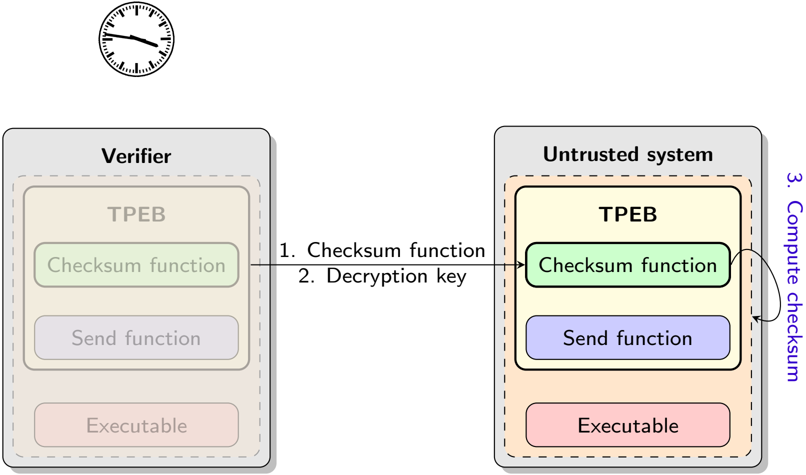
Conqueror: Bullet-proof software-based code attestation



Conqueror: Bullet-proof software-based code attestation



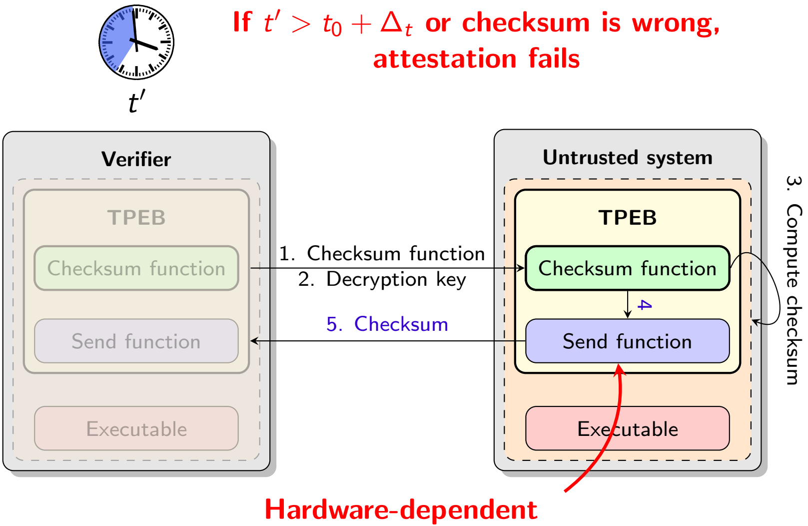
Conqueror: Bullet-proof software-based code attestation



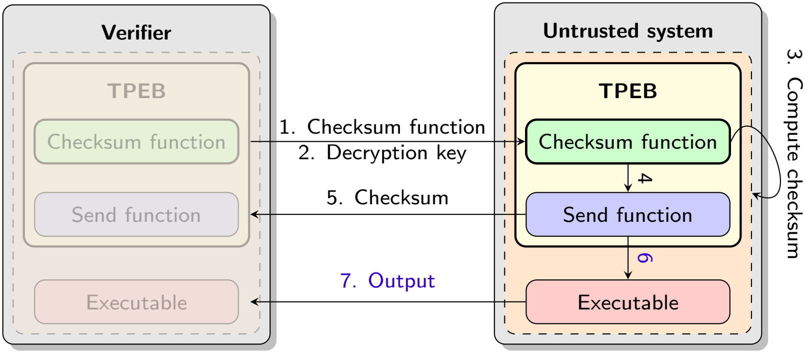
Attests the content of the memory and the execution environment

Conqueror: Bullet-proof software-based code attestation

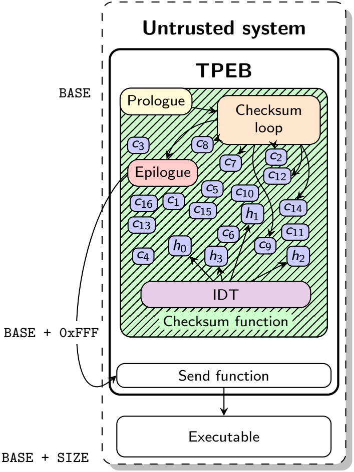
If $t' > t_0 + \Delta_t$ or checksum is wrong, attestation fails



Conqueror: Bullet-proof software-based code attestation

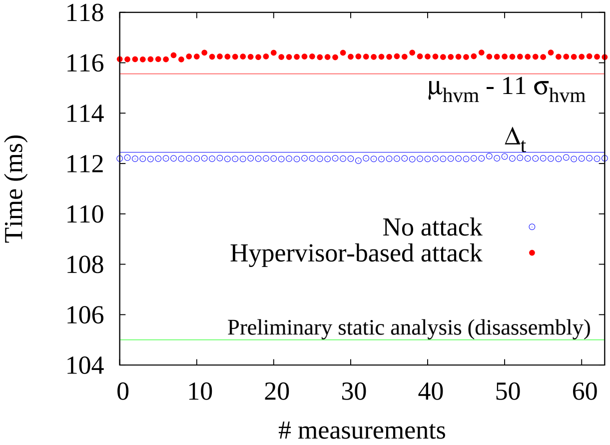


Memory and environment attestation



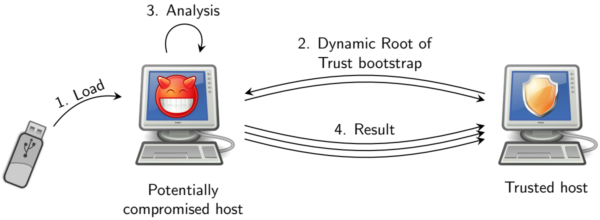
- Checksum computation over the region $[\texttt{BASE}, \texttt{BASE} + \texttt{SIZE})$
- Attest the execution environment
  - Maximum privileges
  - Interrupts disabled
  - No hypervisor

Evaluation: Checksum computation time



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

Live and trustworthy analysis



Problem definition

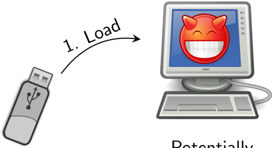
How to perform **live** post-infection (or post-intrusion) analysis, with **no service interruption** ?

HyperSleuth

A framework to perform **live and trustworthy acquisition** of volatile data from commodity production systems

HyperSleuth

A framework to perform **live and trustworthy acquisition** of volatile data from commodity production systems

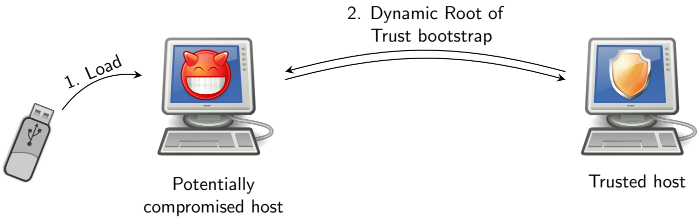


Potentially
compromised host

HyperSleuth is installed on an allegedly compromised target **as the system runs**

HyperSleuth

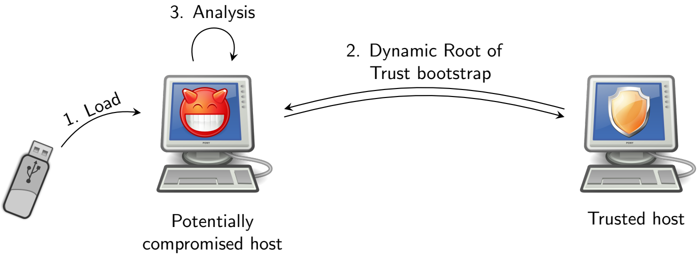
A framework to perform **live and trustworthy acquisition** of volatile data from commodity production systems



The installation of HyperSleuth is **attested** using Conqueror

HyperSleuth

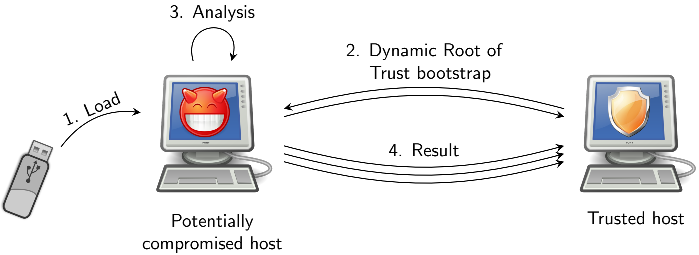
A framework to perform **live and trustworthy acquisition** of volatile data from commodity production systems



The analyzed OS **needs not to be modified** at all, applications continue to run with **no service disruption**

HyperSleuth

A framework to perform **live and trustworthy acquisition** of volatile data from commodity production systems



At the end of the analysis, the results can be sent to the trusted host

How?

1. A tiny hypervisor, based on the previous contribution
2. A secure loader (Conqueror) that installs the hypervisor
    - It verifies the hypervisor's code, data and its environment

Proposed applications

- **Lazy physical memory dumper**
- Lie detector (not discussed in this talk)
- System call tracer (not discussed in this talk)

HyperSleuth: Lazy physical memory dumper

Lazily dumps the content of physical memory

- The CPU is not monopolized
- Processes running in the system are not interrupted

State of *dumped* physical memory $\equiv$ state of physical memory
at the time the dump is requested

HyperSleuth: Lazy physical memory dumper

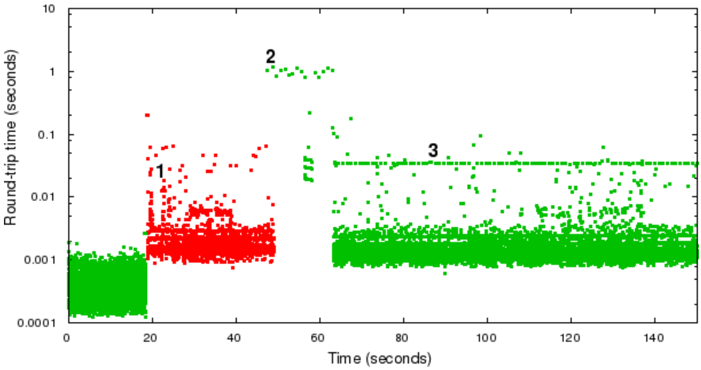
Lazily dumps the content of physical memory

- The CPU is not monopolized
- Processes running in the system are not interrupted

State of *dumped* physical memory ≡ state of physical memory
at the time the dump is requested

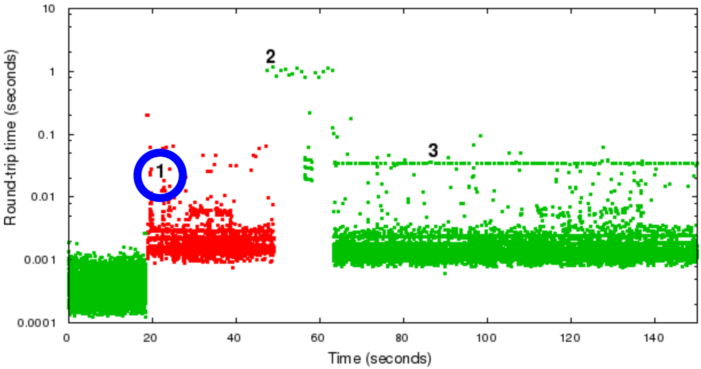
- Dump-on-Write (DOW)
  (i.e., dump the page before it is modified by the guest)
- Dump-on-Idle (DOI)
  (i.e., dump the page when the guest is idle)

Evaluation of the lazy physical memory dumper



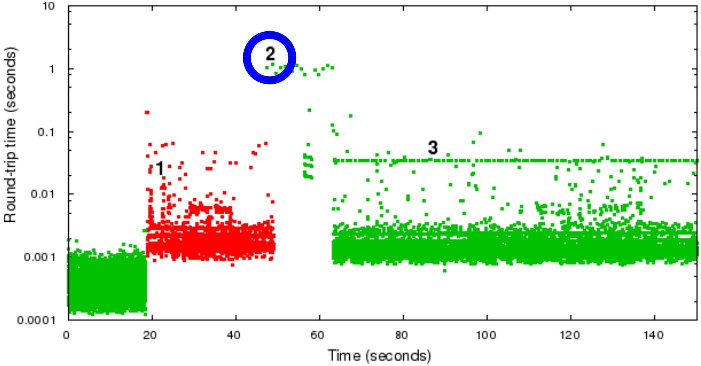
Memory acquisition on a
heavy-loaded DNS server

Evaluation of the lazy physical memory dumper



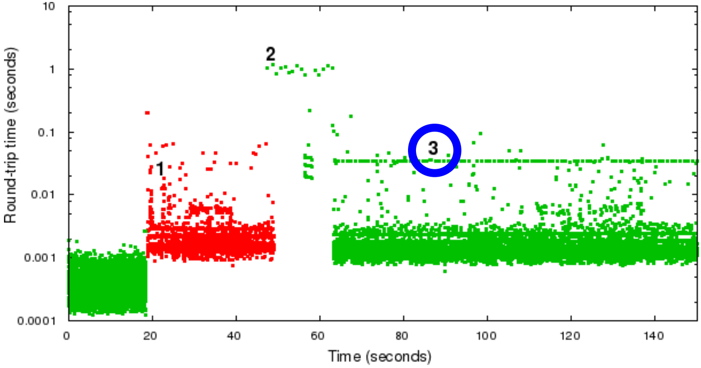
DRT bootstrap and installation
of the VMM

Evaluation of the lazy physical memory dumper



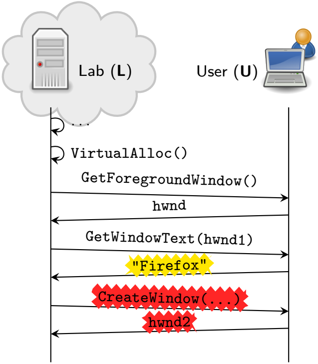
When we started the dump, a lot of frequently accessed pages were dumped

Evaluation of the lazy physical memory dumper



Regular peaks were caused by
periodic dump of non-written pages

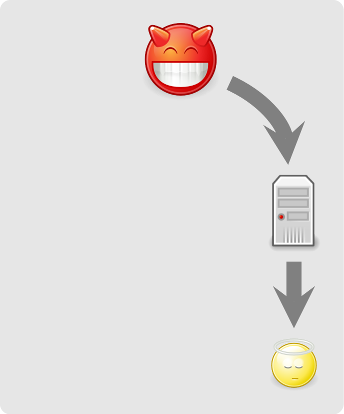
Malware analysis in the cloud



A Framework for Behavior-based Malware Analysis in the Cloud (ICISS 2009)

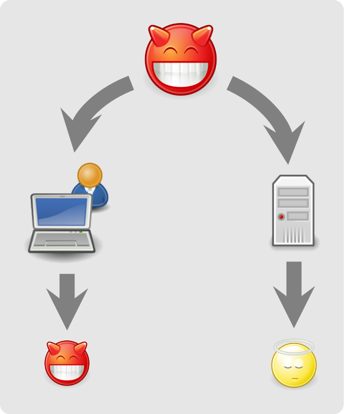
Incompleteness of dynamic behavior-based analysis

The execution environments used in security labs can perform fine-grained analyses, but are **synthetic**
(i.e., not realistic enough to trigger malicious behaviors)

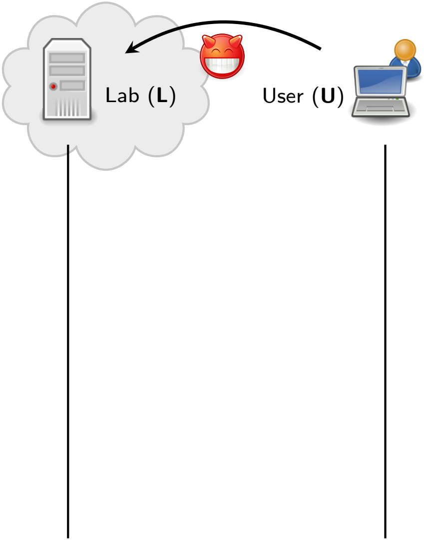


Incompleteness of dynamic behavior-based analysis

End-users' machines lack computational power but provide **realistic** environments
(they are the intended target of the attack)

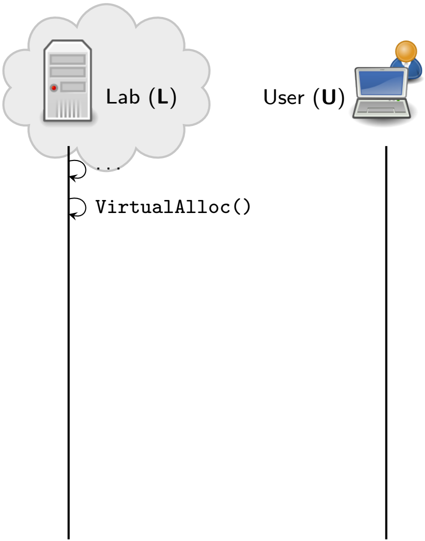


Malware analysis in the cloud



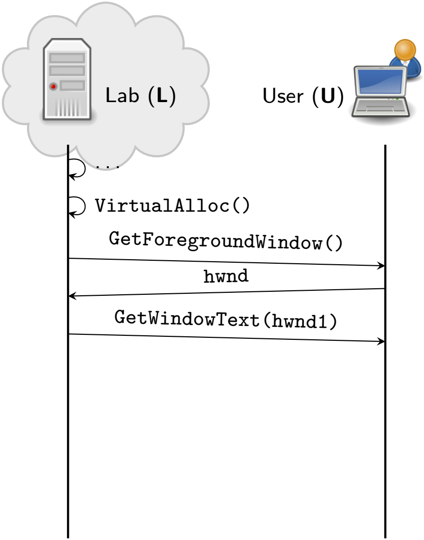
- Execute and analyze in **L**, but force the program to behave as in **U**
- **L** can analyze the behavior of the program in a realistic environment
- **U** benefits from a more fine-grained analysis and one-way isolation

Malware analysis in the cloud



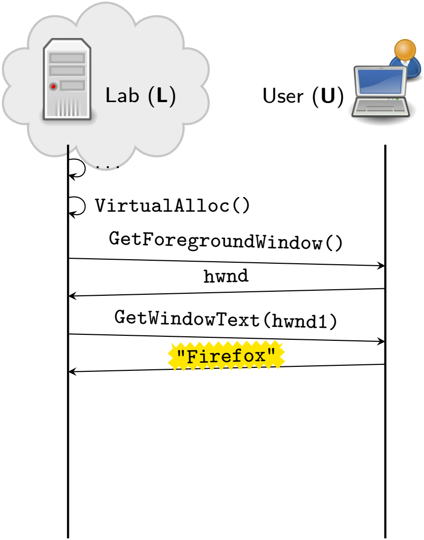
- Intercept all system calls
- Execute system calls that are **not environment dependent** in **L**

Malware analysis in the cloud



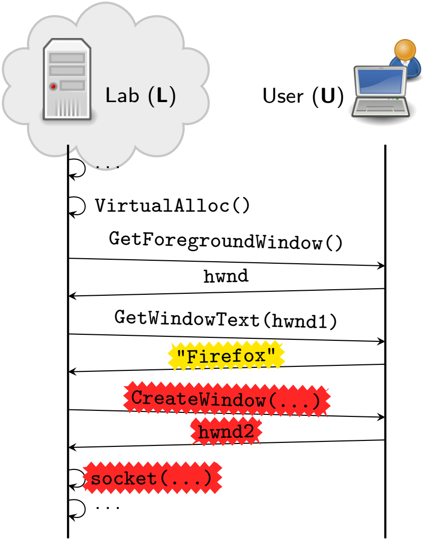
- Intercept all system calls
- Execute system calls that are **not environment dependent** in **L**
- Proxy **environment dependent** system calls to **U** and proxy back the output

Malware analysis in the cloud



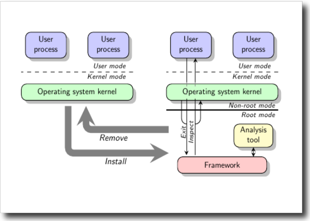
- **U** satisfies the trigger condition of the malicious behavior

Malware analysis in the cloud



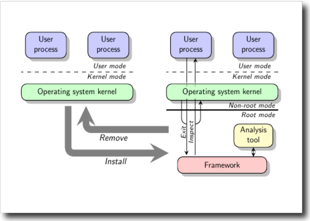
- **U** satisfies the trigger condition of the malicious behavior
- **L** observes the malicious activity

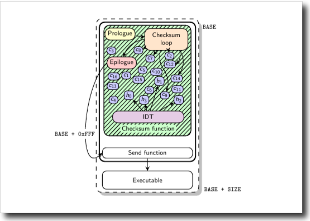
A summary of the contributions



An infrastructure to perform transparent dynamic system-level analyses of deployed production systems

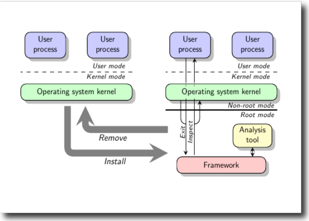
A summary of the contributions

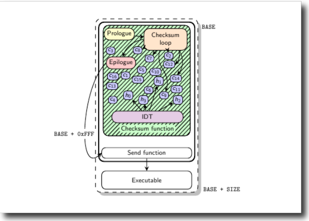


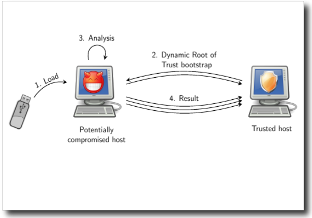


A software-based attestation scheme for tamper-proof code execution on untrusted legacy systems

A summary of the contributions

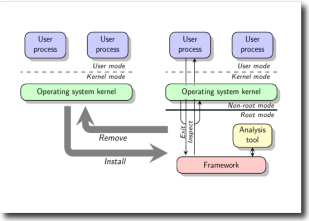


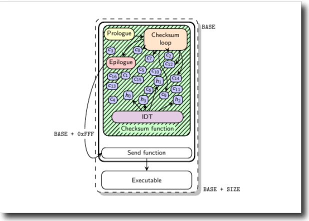


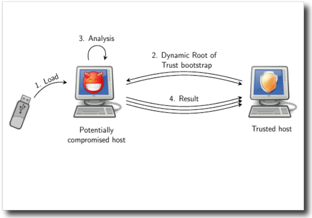


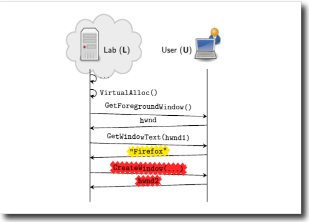
A framework to perform live and trustworthy acquisition of volatile data from commodity production systems

A summary of the contributions









A framework for improving the completeness of behavior-based analysis of suspicious programs

Other contributions

Malware detection & remediation

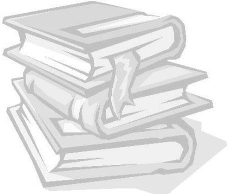
- Automatic generation of remediation procedures for malware infections (USENIX 2010)
- How to automatically generate procedures to detect CPU emulators (WOOT 2010)
- How good are malware detectors at remediating infected systems? (DIMVA 2009)
- FluXOR: detecting and monitoring fast-flux service networks (DIMVA 2008)

Vulnerability analysis

- Surgically returning to randomized lib(c) (ACSAC 2009)
- On race vulnerabilities in web applications (DIMVA 2009)
- A hybrid analysis framework for detecting web application vulnerabilities (SESS 2009)
- A smart fuzzer for x86 executables (SESS 2008)

Software testing

- Testing system virtual machines (ISSTA 2010)
- Differential testing of x86 disassemblers (ISSTA 2010)
- Testing CPU emulators (ISSTA 2009)



Dealing with next-generation malware

Thank you!
Any questions?

Roberto Paleari

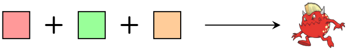
Backup slides

Signature-based detection

Application code

Signature-based detection

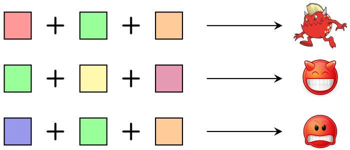
Application code



A signature is a sequence of bytes that identifies a malicious sample

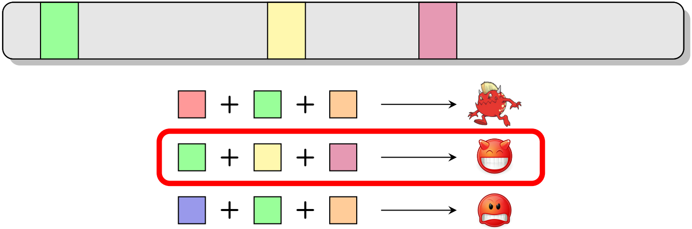
Signature-based detection

Application code



Anti-malware tools are shipped with a database of known signatures

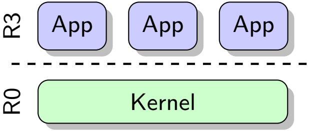
Signature-based detection



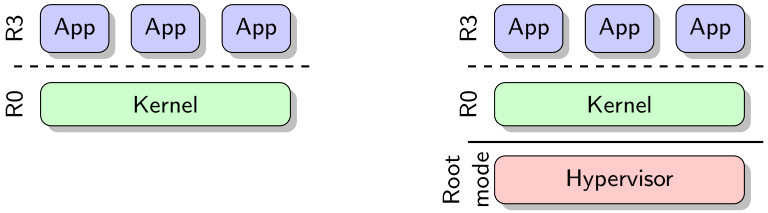
When a signature is found, the application is considered to be infected

Transparent and efficient analysis

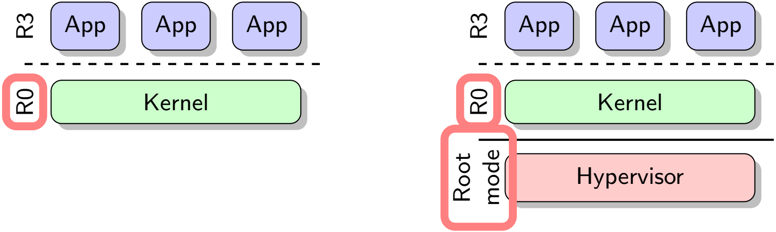
Hardware-assisted virtualization in a nutshell



Hardware-assisted virtualization in a nutshell

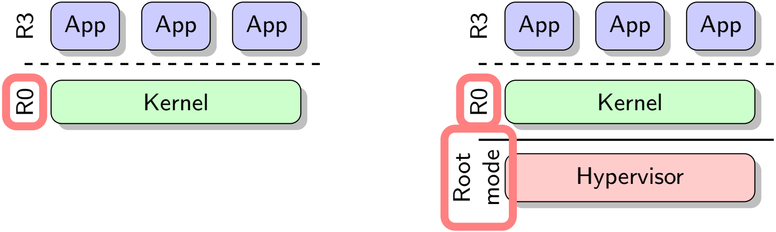


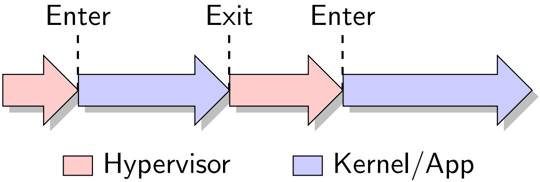
Hardware-assisted virtualization in a nutshell



- The OS needs not to be modified
- Minimal overhead
- The hardware guarantees transparency & isolation
- Available on commodity x86 CPUs

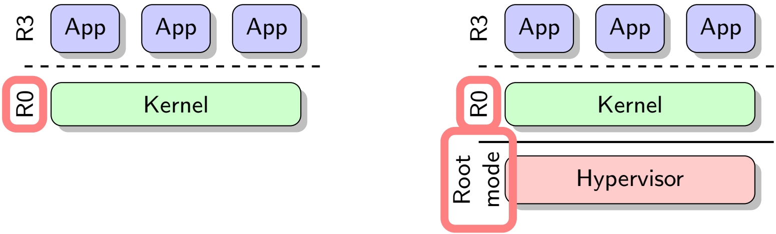
Hardware-assisted virtualization in a nutshell

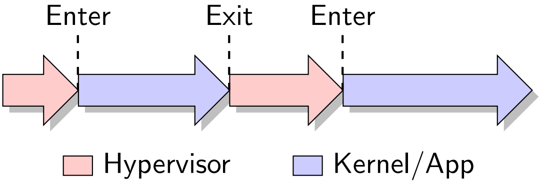




Exit events interrupt the guest and transfer the control of the execution to the hypervisor

Hardware-assisted virtualization in a nutshell





The events that trigger an exit to root mode can be configured **dynamically**

Which events can be intercepted?

- Events cause exits to root mode
- All the events exit **conditionally**
- Conditions are expressed as boolean conditions

  $(\texttt{process\_name} = \text{``}\texttt{notepad.exe}\text{''} \wedge \texttt{syscall\_name} = \text{``}\texttt{NtReadFile}\text{''})$

Which events can be intercepted?

- Events cause exits to root mode
- All the events exit **conditionally**
- Conditions are expressed as boolean conditions
  ($\texttt{process\_name} = \text{``}\texttt{notepad.exe}\text{''} \wedge \texttt{syscall\_name} = \text{``}\texttt{NtReadFile}\text{''}$)

Native events vs high-level events

Native events:

- Traced directly through the hardware
- Very low-level operations (e.g., CPU exception)

High-level events:

- Traced through low-/high-level events
- High-level operations (e.g., Return from function)

A summary of the events

Event	Exit cause	Native exit
`ProcessSwitch`	Change of page table address	√
`Exception`	Exception	√
`Interrupt`	Interrupt	√
`BreakpointHit`	Debug or page fault except.	
`WatchpointHit`	Page fault except.	
`FunctionEntry`	Break on function entry point	
`FunctionExit`	Break on return address	
`SyscallEntry`	Break on syscall entry point	
`SyscallExit`	Break on return address	
`IOOperationPort`	Port read/write	√
`IOOperationMmap`	Watchpoint on device memory	

Software-based code attestation

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
```

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
```

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
```

Gadgets: Instruction and data pointers attestation

- Based on *self-modifying code*
- Prevent *memory copy attacks* (e.g., TLB desynchronization)
- Attest that the VA $\leftrightarrow$ 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
```

Gadgets: Hypervisor detection

- Rich ongoing debate on this topic . . .
- Timing attacks are effective with an external time source
  (i.e., the verifier)
- Execute instructions that *unconditionally* trap to the hypervisor

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

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

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

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

Live and trustworthy analysis

Lazy physical memory dumper

The algorithm

```
switch (VMM exit reason)
  case CR3 write:
     Sync PT and SPT
     for (v = 0; v < sizeof(SPT); v++)
        if (SPT[v].Writable && !DUMPED[SPT[v].PhysicalAddress])
           SPT[v].Writable = 0;
  case Page fault: // 'v' is the faulty address
     if (PT/SPT access)
        Sync PT and SPT and protect SPTEs if necessary
     else if (write access && PT[v].Writable)
        if (!DUMPED[PT[v].PhysicalAddress])
           DUMP(PT[v].PhysicalAddress);
        SPT[v].Writable = DUMPED[PT[v].PhysicalAddress] = 1;
     else
        Pass the exception to the OS
  case Hlt:
     for (p = 0; p < sizeof(DUMPED); p++)
        if (!DUMPED[p])
           DUMP(p); DUMPED[p] = 1;
           break;
```

The VMM intercepts updates of the page table address, page-fault exceptions, and CPU idle loops

Lazy physical memory dumper

The algorithm

```
switch (VMM exit reason)
  case CR3 write:
     Sync PT and SPT
     for (v = 0; v < sizeof(SPT); v++)
        if (SPT[v].Writable && !DUMPED[SPT[v].PhysicalAddress])
           SPT[v].Writable = 0;
  case Page fault: // 'v' is the faulty address
     if (PT/SPT access)
        Sync PT and SPT and protect SPTEs if necessary
     else if (write access && PT[v].Writable)
        if (!DUMPED[PT[v].PhysicalAddress])
           DUMP(PT[v].PhysicalAddress);
        SPT[v].Writable = DUMPED[PT[v].PhysicalAddress] = 1;
     else
        Pass the exception to the OS
  case Hlt:
     for (p = 0; p < sizeof(DUMPED); p++)
        if (!DUMPED[p])
           DUMP(p); DUMPED[p] = 1;
           break;
```

During a context switch (CR3 update) the algorithm grants **read-only** permissions to physical not yet dumped pages

Lazy physical memory dumper

The algorithm

```
switch (VMM exit reason)
  case CR3 write:
     Sync PT and SPT
     for (v = 0; v < sizeof(SPT); v++)
        if (SPT[v].Writable && !DUMPED[SPT[v].PhysicalAddress])
           SPT[v].Writable = 0;
  case Page fault: // 'v' is the faulty address
     if (PT/SPT access)
        Sync PT and SPT and protect SPTEs if necessary
     else if (write access && PT[v].Writable)
        if (!DUMPED[PT[v].PhysicalAddress])
           DUMP(PT[v].PhysicalAddress);
        SPT[v].Writable = DUMPED[PT[v].PhysicalAddress] = 1;
     else
        Pass the exception to the OS
  case Hlt:
     for (p = 0; p < sizeof(DUMPED); p++)
        if (!DUMPED[p])
           DUMP(p); DUMPED[p] = 1;
           break;
```

Our write protection is reinforced after every update of the page tables

Lazy physical memory dumper

The algorithm

```
switch (VMM exit reason)
  case CR3 write:
     Sync PT and SPT
     for (v = 0; v < sizeof(SPT); v++)
        if (SPT[v].Writable && !DUMPED[SPT[v].PhysicalAddress])
           SPT[v].Writable = 0;
  case Page fault: // 'v' is the faulty address
     if (PT/SPT access)
        Sync PT and SPT and protect SPTEs if necessary
     else if (write access && PT[v].Writable)
        if (!DUMPED[PT[v].PhysicalAddress])
           DUMP(PT[v].PhysicalAddress);
        SPT[v].Writable = DUMPED[PT[v].PhysicalAddress] = 1;
     else
        Pass the exception to the OS
  case Hlt:
     for (p = 0; p < sizeof(DUMPED); p++)
        if (!DUMPED[p])
           DUMP(p); DUMPED[p] = 1;
           break;
```

Write accesses to pages not yet dumped trigger **page fault** exceptions, and pages are dumped before being modified (DOW)

Lazy physical memory dumper

The algorithm

```
switch (VMM exit reason)
  case CR3 write:
     Sync PT and SPT
     for (v = 0; v < sizeof(SPT); v++)
        if (SPT[v].Writable && !DUMPED[SPT[v].PhysicalAddress])
           SPT[v].Writable = 0;
  case Page fault: // 'v' is the faulty address
     if (PT/SPT access)
        Sync PT and SPT and protect SPTEs if necessary
     else if (write access && PT[v].Writable)
        if (!DUMPED[PT[v].PhysicalAddress])
           DUMP(PT[v].PhysicalAddress);
        SPT[v].Writable = DUMPED[PT[v].PhysicalAddress] = 1;
     else
        Pass the exception to the OS
  case Hlt:
     for (p = 0; p < sizeof(DUMPED); p++)
        if (!DUMPED[p])
           DUMP(p); DUMPED[p] = 1;
           break;
```

To guarantee termination, pending pages are dumped on CPU idle loops

Lie detector

- Kernel-level malware insidious and dangerous
  - Operate at a very high privilege level
  - Able to hide any resource an attacker wants to protect (e.g., processes, network communications, files)
- Different techniques to force the OS to lie about its state
- How can we disguise such liars?
  - Retrieve $\mathcal{S}_{guest}$, the state perceived by the (guest) system
  - Retrieve $\mathcal{S}_{VMM}$, the state perceived by the VMM (OS-aware inspection)
  - $\mathcal{S}_{guest} = \mathcal{S}_{VMM}$?

Lie detector

Evaluation

Sample	Characteristics	Detected?
FU	DKOM	✓
FUTo	DKOM	✓
HaxDoor	DKOM, SSDT hooking, API hooking	✓
HE4Hook	SSDT hooking	✓
NtIllusion	DLL injection	✓
NucleRoot	API hooking	✓
Sinowal	MBR infection, Run-time patching	✓
Smiscer	DKOM, Run-time patching	✓
TDL3	DKOM, Run-time patching	✓

Lie detector
Evaluation

Sample	Characteristics	Detected?
FU	DKOM	✓
FUTo	DKOM	✓
HaxDoor	DKOM, SSDT hooking, API hooking	✓
HE4Hook	SSDT hooking	✓
NtIllusion	DLL injection	✓
NucleRoot	API hooking	✓
Sinowal	MBR infection, Run-time patching	✓
Smiscer	DKOM, Run-time patching	✓
TDL3	DKOM, Run-time patching	✓

FUTo leverages DKOM to hide malicious resources. We scan Windows' internal structures that must be left intact to preserve system functionalities

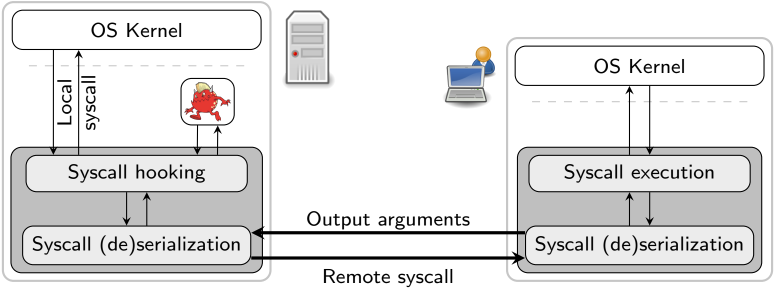
Lie detector
Evaluation

Sample	Characteristics	Detected?
FU	DKOM	✓
FUTo	DKOM	✓
HaxDoor	DKOM, SSDT hooking, API hooking	✓
HE4Hook	SSDT hooking	✓
NtIllusion	DLL injection	✓
NucleRoot	API hooking	✓
Sinowal	MBR infection, Run-time patching	✓
Smiscer	DKOM, Run-time patching	✓
TDL3	DKOM, Run-time patching	✓

HaxDoor hooks system calls and filters their result. We observed hidden registry keys were missing from the untrusted view.

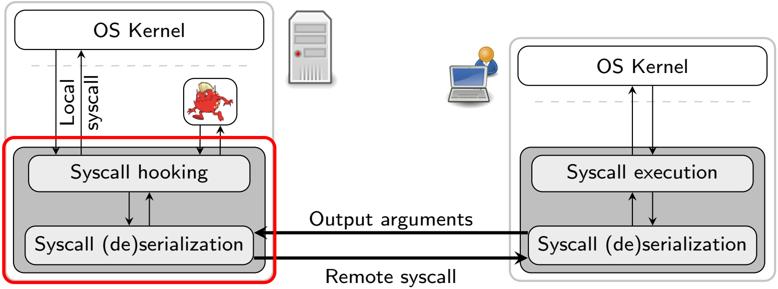
Malware analysis in the cloud

A glimpse at the implementation



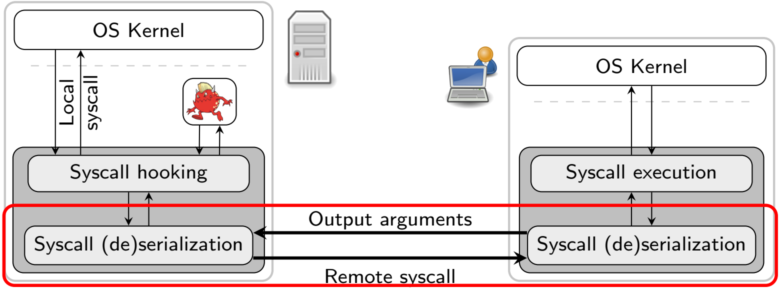
Prototype implementation for Microsoft Windows XP and Linux

A glimpse at the implementation



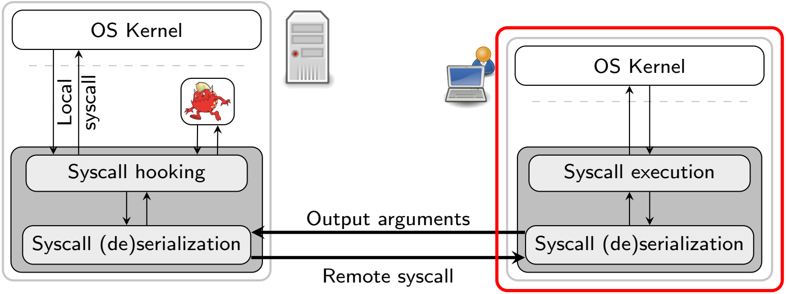
Intercept all system calls (through user-space hooking)
and analyze the resources they manipulate

A glimpse at the implementation



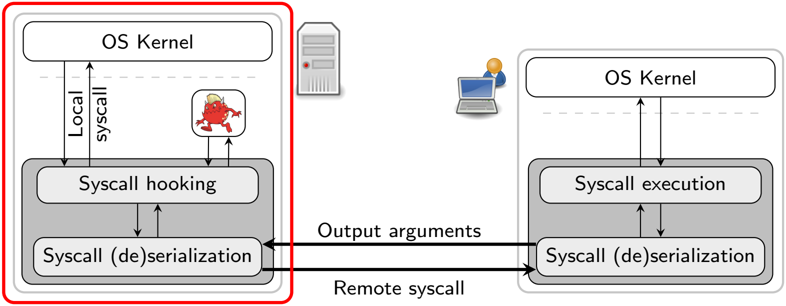
Serialize environment dependent system calls and arguments and transmit them over the network

A glimpse at the implementation



End-user's system is protected using **one-way isolation**

A glimpse at the implementation



Labs can devote all their computation power for the analysis and can exploit hardware features, in combination with recent advances in research

Evaluation: Correctness and performance overhead

Correct execution of benign programs

- Successfully executed multiple real-world benign programs
- No interference with the correct execution of programs
- Transparently accessed all resources residing on a remote host

Program	**Action**	**Local**	**Remote**
ClamAV	Scan (remote) files with (remote) signatures	166,539	1,238
Eudora	Access and query (remote) address book	1,418,162	11,411
Gzip	Compress (remote) files	19,715	93
MS IE	Open a (remote) HTML document	1,263,385	10,260
MS Paint	Browse, open, and edit (remote) pictures	1,177,818	9,708
Netcat	Transfer (remote) files to another host	16,007	93
Notepad	Browse, open, and edit (remote) text files	929,191	7,598
RegEdit	Browse, view, and edit (remote) registry keys	1,573,995	13,697
Task Mgr.	List (remote) running processes	33,339	241
WinRAR	Decompress (remote) files	71,195	572

Evaluation: Relative code coverage increase with malware