Measuring the Semantic Integrity of a Process Self

Ph.D. Thesis Defense

Daniele Sgandurra

Università di Pisa

24 June 2010
Outline

Measuring the Semantic Integrity
Virtualization-Based Security
Description of the Process Self
Run-Time Architecture
Remote Attestation of Semantic Integrity
Code Obfuscation in a Virtual Environment
Trusted Overlays of Virtual Communities
Main Contributions of the Thesis

- Definition and evaluation of PsycoTrace, a security framework to protect a process from attacks against the process self:
  - measuring the semantic integrity;
  - static analysis and run-time monitoring;
  - complete transparency of the proposed framework.

- Three applications of the proposed framework targeted at cloud computing:
  - remote attestation;
  - code obfuscation;
  - secure sharing of cloud resources.
Part I

Background
Attacks

- **Attacks against user-level processes:**
  - the attacker injects some code into a process;
  - the attacker diverges the original control-flow to execute the injected code.

- **Attacks against the kernel:**
  - modify some kernel functionalities;
  - final step of a complex attack (requires root privileges);
  - modify the kernel behavior to hide any sign of the attack.
Process Self

- The properties of a process that determine its run-time behavior define the process self:
  - the process self can be approximated through static analysis.
- We assume that if the process current behavior deviates from the process self then the process code has been altered by an attack.
- Measuring the semantic integrity: the act of approximating the process self and of monitoring the actual process behavior to assure that it is coherent with the process self.
- $P$ is a generic process that we want to protect.
- $\text{Self}(P)$ refers to the process self of $P$.
- $\text{SourceCode}(P)$ is the source code of the program executed by $P$. 
Process Self

$\text{process self} 
\begin{array}{c} \text{program} \\ \text{current behaviour} \\ \text{run-time monitoring} \\ \text{monitor trace of system calls} \\ \text{check invariants} \end{array} 
\begin{array}{c} \text{exec(program);} \\ \text{extract the properties} \\ \text{analyze the code} \\ \text{compile the code} \end{array} 
\begin{array}{c} \text{program} \\ \text{source code} \end{array} 
\begin{array}{c} \text{static analysis} \\ \text{measuring the semantic integrity} \end{array} 
\begin{array}{c} \text{inherit the self} \end{array} 
\begin{array}{c} \text{compare} \end{array} 
\begin{array}{c} \text{SourceCode(P)} \end{array}
Alternative Descriptions of the Process Self

- **Hashing or Memory Invariants**: memory invariants to be evaluated anytime \( P \) issues a given system call;
- **Forbidden Calls**: the set of system calls that \( P \) cannot issue;
- **Forbidden Parameters**: the set of system calls that \( P \) cannot issue and assertions on parameters it cannot transmit to a call;
- **Allowed Calls**: the set of system calls that \( P \) can issue and assertions on the parameters;
- **Enriched Traces**: describe the sequence of system calls that \( P \) issues in one execution; each call may be coupled with a memory assertion.
Enriched Traces

- A set of enriched traces fully describes alternative legal behaviors of $P$.
- To support this strategy, PsycoTrace static tools map $SourceCode(P)$ into a description of the $Self(P)$ that is $CFG(P)$ and $IT(P)$:
  - $CFG(P)$ is context-free grammar that defines the system call traces that $P$ may issue during its execution.
  - $IT(P)$ is a set of invariants $\{I(P, 1), \ldots, I(P, n)\}$, each associated with a program point $i$ where $P$ invokes a system call.
Virtualization

- **Virtual Machine Monitor (VMM)**: manages Virtual Machines (VMs), i.e. execution environments that emulate, at software, the behavior of the underlying physical machine.

- A real machine can support several VMs, each with a distinct OS.
Virtual Machine Introspection

- Virtual Machine Introspection (VMI) enables a privileged VM, or Introspection VM (I-VM), to retrieve critical data structures in the memory of a Monitored VM (Mon-VM) at the kernel or at the user-level.

- An I-VM can analyze the state of the processes/kernel on a Mon-VM at the HW/FW level, without introducing additional HW/FW units.

- Introspection is applied at a lower level than the one an attacker can gain and it is very complex to elude it.
Introspection VM

Virtual Machine Monitor

Monitored VM

Kernel

User-Level Processes

Virtualization-Based Security
Advantages of Virtual Machine Introspection

- **Full visibility** of the system running inside the Mon-VM: the I-VM can access every Mon-VM component, such as the main memory or the processor’s registers.

- **More robustness**: the I-VM is isolated from the Mon-VM.

- **Transparency**: the security checks can be implemented without modifying the software on the Mon-VM and they are almost invisible.
Part II

Principles and Implementation
Static Tools

PsycoTrace static tools analyze SourceCode($P$) to approximate the process self:

1. one tool implements Grammar Generating Algorithm, an algorithm that builds $CFG(P)$, the grammar that describes the traces of $P$;
2. the Assertion Generator generates the set of invariants $IT(P)$.

A preliminary step of the static analysis builds Abstract Syntax Tree of $P$ ($AST(P)$), which is exploited by both tools.
Grammar Generating Algorithm

- PsycoTrace static tools generate $CFG(P)$ by applying grammar generating algorithm (GGA) while traversing $AST(P)$.

- $CFG(P)$ is a tuple $< T, F, S, R >$:
  - $T$ is a set of terminal symbols with one symbol for each distinct system call in $SourceCode(P)$;
  - $F$ is a set of non-terminal symbols, one for each function defined in $SourceCode(P)$; each symbol corresponds to a subset of $T$.
  - $S \in F$ is the starting symbol, which corresponds to `main`;
  - $R$ is the set of production rules $X \rightarrow \beta$, where $X$ is a non-terminal symbol and $\beta$ a sequence of terminal and not-terminal symbols.
Grammar Generating Algorithm Rules

- GGA analyzes $AST(P)$ and for each function $\text{fun}$ defined in $SourceCode(P)$ it inserts into $F$ a new non-terminal symbol $\text{FUN}$ and a new rule $R_{\text{new}}$ into $R$, where $\text{FUN}$ is the left-hand-side of $R_{\text{new}}$.

- To generate the right-hand side of the rule, GGA linearly scans the definition of $\text{fun}$ in $SourceCode(P)$.

- Distinct production rules may be generated, according to the type of statements met by GGA in the body of $\text{fun}$.

- For each statement, GGA generates a new rule and adds a new symbol to the right-hand side of $R_{\text{new}}$.

- In this way, $CFG(P)$ represents the system calls that $\text{fun}$ can invoke and the ordering among the invocations in the body in $\text{fun}$.
18/65 Measuring the Semantic Integrity of a Process Self

Description of the Process Self

```plaintext
1  f () {
2      open ();
3      read ();
4      g ();
5      close ();
6  }
7
8  g () {
9      getpid ();
10  }
```

```plaintext
\( \langle F \rangle \rightarrow \) open read \( \langle G \rangle \) close;
\( \langle G \rangle \rightarrow \) getpid;
```

```plaintext
1  f () {
2      open ();
3      if (x)
4          read ();
5  }
```

```plaintext
\( \langle F \rangle \rightarrow \) open \( \langle ST_1 \rangle \);
\( \langle ST_1 \rangle \rightarrow \) read \( \mid \epsilon \);
```

```plaintext
1  f () {
2      open ();
3      if (x)
4          read ();
5      else
6          close ();
7  }
```

```plaintext
\( \langle F \rangle \rightarrow \) open \( \langle IFEL_1 \rangle \);
\( \langle IFEL_1 \rangle \rightarrow \) \( \langle STIF_2 \rangle \mid \)
\( \langle ELSE_3 \rangle \);
\( \langle STIF_2 \rangle \rightarrow \) read;
\( \langle ELSE_3 \rangle \rightarrow \) close;
```
19/65        Measuring the Semantic Integrity of a Process Self

— Description of the Process Self

```java
1  f()
2    open();
3    while(x)
4      read();
5  }
```

```latex
\langle F \rangle \rightarrow \text{open } \langle \text{WHILE}_1 \rangle;
\langle \text{WHILE}_1 \rangle \rightarrow \langle \text{WHRE}_2 \rangle
\langle \text{WHRE}_3 \rangle | \epsilon;
\langle \text{WHRE}_2 \rangle \rightarrow \text{read};
```

```java
1  f()
2    open();
3    read();
4    if(execl(...) != -1)
5      f();
6    close();
7  }
```

```latex
\langle F \rangle \rightarrow \text{open read } \langle \text{STIF}_1 \rangle
\text{close};
\langle \text{STIF}_1 \rangle \rightarrow \text{execl } \langle F \rangle | \epsilon;
```
SourceCode(P) Example

```c
int main(int argc, char **argv) {
    char *filename;
    int fd;
    filename = argv[1];
    fd = fork();
    if (fork() == 0) {
        if (execl(filename, NULL) == -1) {
            printf("ERROR!\n");
        } else {
            strcat(filename, " .execl ");
        }
    } else {
        ... 
    }
    if (argc > 2) {
        filename = argv[2];
        strcat(filename, " .execl2 ");
    }
    execl(filename, NULL);
}
```
### CFG($P$) Example

\[
\langle \text{MAIN}_P \rangle \rightarrow \langle \text{EXPR}_0 \rangle \langle \text{EXPR}_1 \rangle \langle \text{IFEL}_2 \rangle \langle \text{STIF}_{11} \rangle \langle \text{EXPR}_{14} \rangle ;
\]
\[
\langle \text{EXPR}_0 \rangle \rightarrow /*\text{empty}*/ ;
\]
\[
\langle \text{EXPR}_1 \rangle \rightarrow \text{fork} ;
\]
\[
\langle \text{IFEL}_2 \rangle \rightarrow \langle \text{STIF}_3 \rangle | \langle \text{ELSE}_4 \rangle ;
\]
\[
\langle \text{ELSE}_4 \rangle \rightarrow \text{fork} \langle \text{EXPR}_{10} \rangle ;
\]
\[
\langle \text{STIF}_3 \rangle \rightarrow \text{fork} \langle \text{IFEL}_5 \rangle ;
\]
\[
\langle \text{IFEL}_5 \rangle \rightarrow \langle \text{STIF}_6 \rangle | \langle \text{ELSE}_7 \rangle ;
\]
\[
\langle \text{ELSE}_7 \rangle \rightarrow \text{execl} \langle \text{EXPR}_9 \rangle ;
\]
\[
\langle \text{STIF}_6 \rangle \rightarrow \text{execl} \langle \text{IFEL}_8 \rangle ;
\]
\[
\langle \text{EXPR}_8 \rangle \rightarrow \langle \text{PRINTF}_p \rangle ;
\]
\[
\langle \text{PRINTF}_p \rangle \rightarrow \ldots ;
\]
\[
\langle \text{EXPR}_9 \rangle \rightarrow \langle \text{STRCAT}_{p_1} \rangle ;
\]
\[
\langle \text{STRCAT}_{p_1} \rangle \rightarrow \ldots ;
\]
\[
\langle \text{EXPR}_{10} \rangle \rightarrow \ldots ;
\]
\[
\langle \text{STIF}_{11} \rangle \rightarrow /*\text{empty}*/ | \langle \text{EXPR}_{12} \rangle \langle \text{EXPR}_{13} \rangle
\]
\[
\langle \text{EXPR}_{12} \rangle \rightarrow /*\text{empty}*/ ;
\]
\[
\langle \text{EXPR}_{13} \rangle \rightarrow \langle \text{STRCAT}_{p_1} \rangle ;
\]
\[
\langle \text{EXPR}_{14} \rangle \rightarrow \text{execl} ;
\]
The Assertion Generator traverses $AST(P)$ and analyzes the variables, functions and language statements to build the invariant table ($IT(P)$).

To simplify the analysis, we restrict to:

- **integer variables**: only files and socket descriptors to express relations among these variables and the system calls;
- **string variables**: in case of arrays of char statically declared, functions to manipulate strings are treated like assignments;
- **struct members**: only integer or string type field.
Classes of Assertion

Any assertion is the composition of any of the followings:

1. **Parameters assertions.** They express data-flow relations among parameters of distinct calls, e.g. the file descriptor in a read call is the result of a previous open call.

2. **File Assertions.** To prevent symlink and race condition attacks, they check, as an example, that the real file-name corresponding to the file descriptor belongs to a known directory.

3. **Buffer length assertions.** They check that the length of the string passed to a vulnerable function is not larger than the local buffer to hold it.

4. **Conditional statements assertions.** They prevent problems due to impossible paths by relating a system call and the expression in the guard of a conditional statement.
SourceCode$(P)$ Example

```c
/*
 * invariant generation example
 */

int main(int argc, char **argv) {
    char path1[10];
    char path2[10] = "B";
    int test, fd;
    if (atoi(argv[1]) == 0) {
        test = 2;
        strcpy(path2, "D");
        strcpy(path1, "D");
        fd = open("file_path", "RW");
    }
    if (atoi(argv[1]) > 5) {
        test = 1;
        strcpy(path1, "A");
        printf("%s", path1);
        write(fd, path1, sizeof(path1));
    }
    else {
        test = 0;
        execl(path2, "");
        printf("%s", path1);
    }
}```
<table>
<thead>
<tr>
<th>Rule</th>
<th>IN Variant</th>
<th>RT</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRM_FUNC@main</td>
<td>argc = {}</td>
<td>RT</td>
<td>5</td>
</tr>
<tr>
<td>PRM_FUNC@main</td>
<td>argv = {}</td>
<td>RT</td>
<td>5</td>
</tr>
<tr>
<td>main_stringsSYS</td>
<td>path1 = {}</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>STIF_0</td>
<td>path1 = {&quot;D&quot;}</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>STIF_6</td>
<td>path1 = {&quot;D&quot;,&quot;A&quot;}</td>
<td>-</td>
<td>17</td>
</tr>
<tr>
<td>STIF_6</td>
<td>path1 = {&quot;D&quot;,&quot;A&quot;}</td>
<td>-</td>
<td>18</td>
</tr>
<tr>
<td>ELSE_7</td>
<td>path1 = {&quot;D&quot;,&quot;A&quot;}</td>
<td>-</td>
<td>24</td>
</tr>
<tr>
<td>main_stringsSYS</td>
<td>path2 = &quot;B&quot;</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>STIF_0</td>
<td>path2 = {&quot;D&quot;}</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>ELSE_7</td>
<td>path2 = {&quot;D&quot;}</td>
<td>-</td>
<td>23</td>
</tr>
<tr>
<td>main_stringsSYS</td>
<td>test = {}</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>STIF_0</td>
<td>test = {2,1}</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>STIF_6</td>
<td>test = {2,1,0}</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td>ELSE_7</td>
<td>test = {2,1,0}</td>
<td>-</td>
<td>22</td>
</tr>
<tr>
<td>main_stringsSYS</td>
<td>fd = {}</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>STIF_0</td>
<td>fd = {&quot;open&quot;}</td>
<td>RT</td>
<td>13</td>
</tr>
<tr>
<td>STIF_6</td>
<td>fd = {&quot;open&quot;}</td>
<td>RT</td>
<td>19</td>
</tr>
<tr>
<td>EXPR_2</td>
<td>path2_0@strcpy0 = &quot;B&quot;</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>EXPR_2</td>
<td>PRM_1@strcpy0 = &quot;D&quot;</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>EXPR_3</td>
<td>path1_0@strcpy1 = {}</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>EXPR_3</td>
<td>PRM_1@strcpy1 = &quot;D&quot;</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>EXPR_4</td>
<td>PRM_0@open2 = &quot;file_path&quot;</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>EXPR_4</td>
<td>PRM_1@open2 = &quot;RW&quot;</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>EXPR_8</td>
<td>path1_0@strcpy4 = {&quot;D&quot;}</td>
<td>-</td>
<td>17</td>
</tr>
<tr>
<td>EXPR_8</td>
<td>PRM_1@strcpy4 = &quot;A&quot;</td>
<td>-</td>
<td>17</td>
</tr>
<tr>
<td>EXPR_9</td>
<td>PRM_0@printf5 = &quot;%s&quot;</td>
<td>-</td>
<td>18</td>
</tr>
<tr>
<td>EXPR_9</td>
<td>path1_1@printf5 = &quot;A&quot;</td>
<td>-</td>
<td>18</td>
</tr>
<tr>
<td>EXPR_10</td>
<td>fd_0@write5 = {OPEN_13}</td>
<td>RT</td>
<td>19</td>
</tr>
<tr>
<td>EXPR_11</td>
<td>path1_1@write6 = &quot;A&quot;</td>
<td>-</td>
<td>19</td>
</tr>
<tr>
<td>EXPR_12</td>
<td>path2_0@execl7 = &quot;B&quot;,&quot;D&quot;</td>
<td>-</td>
<td>23</td>
</tr>
<tr>
<td>EXPR_13</td>
<td>PRM_1@execl7 = &quot;&quot;</td>
<td>-</td>
<td>23</td>
</tr>
<tr>
<td>EXPR_13</td>
<td>PRM_0@printf8 = &quot;%s&quot;</td>
<td>-</td>
<td>24</td>
</tr>
</tbody>
</table>
Run-Time Components

- The run-time architecture consists of two virtual machines:
  1. the monitored VM (Mon-VM), i.e. the VM executing $P$;
  2. the introspection VM (I-VM), i.e. the VM monitoring $P$ through virtual machine introspection.

- The I-VM can access each component of the Mon-VM to inspect its running state both to:
  - check the process self of $P$;
  - verify the kernel integrity.
Checking the Process Self at Run-Time

- Provided that the integrity of the Mon-VM kernel is assured, the run-time support has to guarantee that the self of $P$ is not altered.
- The HiMod in the Mon-VM is a module that hijacks the system calls:
  - not needed anymore with VT extensions: full transparency.
- Every time $P$ invokes a syscall, HiMod notifies the Analyst on the I-VM.
The Analyst

The Analyst in the I-VM verifies the integrity of the self of $P$ through:

- **Lexical Analyzer**: it verifies that the system call that $P$ wants to issue belongs to the set of system calls returned by the static analysis of $\text{SourceCode}(P)$;
- **Parser**: it checks that the current trace of system calls issued by $P$ is coherent with $\text{CFG}(P)$, i.e. it is a prefix of a word allowed by $\text{CFG}(P)$;
- **Assertion Checker**: it checks whether the invariant coupled with the current system-call holds.
The Analyst

Monitored VM (MON-VM)
- traced process
- syscall();
- Kernel
- HiMod
- syscall hijack module
- original syscall(s)
- Variables
- Addresses
- 0xB...
- VCPU-C

Introspection VM (I-VM)
- Memory Introspection
- Map
- Tracing/VCPU-Introspection
- 0xB...
- Intros. Library
- ANALYST
- ASSERTION CHECKER
- invariants
- CFG(P)
- PARSER+
  LEX. ANAL.

Event channel (Synchronization)

Xen Virtual Machine Monitor
The I-VM runs an **Assertion Checker** that evaluates invariants on $P$;

Access to the variables of $P$ and the CPU of the Mon-VM is implemented through an **Introspection Library**;

Every time $P$ issues a system call the I-VM:

(i) retrieves the **system call number** and the value of its parameter;
(ii) determines the invariant coupled with the issued system call;
(iii) retrieves the **values of the variables** that the invariant refers to;
(iv) evaluates the invariant and:
   - kills $P$ if the invariant is false;
   - otherwise it resumes the execution of $P$.  

**Assertion Checker**
Invariant Evaluation

Monitored VM (MON-VM)

memory & VCPU-context introspection

1) VCPU intr("kernel_sp") \rightarrow 0xc762fff8
2) map(0xc762fff8)
3) read "esp" value \rightarrow 0xbfadf808
4) map(0xbfadf808)
5) read "ret" value \rightarrow 0x804859d
6) map(0xbfadf858): "i" in invariant set
7) read "i" value \rightarrow 5
8) evaluate invariant: (i==5)?

Introspection VM (I-VM)

Invariant Evaluation

[XenStore/Event channel (synchronization)]

Xen Virtual Machine Monitor
Performance Results

- The average time to map one page of the process into the Assertion Checker address space is about 50\(\mu\)secs:
  - at least three pages (kernel stack, user stack, variable): 150\(\mu\)secs.
- By exploiting a software TLB, each access requires 20\(\mu\)secs, provided that variables are stored in the same page;
  - 60\(\mu\)secs overhead for each evaluation.
- Taking into account the rate of system call invocations, the average execution time overhead is at most 20%.
Kernel Integrity

- Periodically, the I-VM checks the kernel integrity.
- Implemented functions:
  - Detecting Kernel Modifications
  - Running Processes Checker
  - Open Files Checker
  - Loaded Modules Authenticator
  - Promiscuous Mode Checker
  - Anti-spoofing
- Context-Agent.
The I-VM can inject a context-agent inside a Mon-VM to bridge the semantic-gap.

The I-VM also protects the context-agent from modifications by any software running in the Mon-VM.

The I-VM has to provide a trusted subset of the Mon-VM kernel through the integrity functions:

- the context-agent relies on the integrity of some key-components on the Mon-VM kernel.
Context-Agent
Part III

Applications of the Proposed Approach
Virtual machine Integrity Measurement System

- Virtual machine Integrity Measurement System (VIMS) runs on a cloud architecture to continuously attest the integrity of cloud nodes by applying alternative integrity measurements.

- VIMS protects the integrity of a node by defining both a start-up attestation and a continuous monitoring that are applied when a node joins the overlay and as long as the node belongs to the overlay.

- To implement the corresponding measurements, VIMS applies PsycoTrace static tools and extends the run-time ones.
Virtual machine Integrity Measurement System

- Each node of the overlay runs two VMs:
  - the Assurance VM (A-VM), an extension of the I-VM;
  - the Mon-VM.
- The A-VMs cooperate to apply measurements on behalf of the overlay by accessing the live state of the Mon-VM.
- Trust in VIMS measurements requires the correct configuration of both the A-VM and the underlying VMM and it is guaranteed by measurements and controls of a Trusted Platform Module (TPM).
- One of the first frameworks that merges TPM and cloud computing.
VIMS Architecture

1. Mon-VMreq wants to join the overlay ove
2. A-VMove opens a control channel with A-VMreq
3. A-VMreq applies PsycoTrace run-time integrity checks on Mon-VMreq's running software
4. A-VMreq sends measurements to A-VMove
5. Nreq accesses the overlay ove
Overview

- Obfuscation targeted at cloud environments to separate the obfuscated program from the deobfuscator.

- The proposed strategy:
  1. decomposes a program into fragments;
  2. applies the encryption of fragments;
  3. randomizes the control-flow across fragments.
At anytime, *exactly one program fragment*, the current one, is stored in clear on the Mon-VM, whereas any other fragment is encrypted.

As soon as the execution of the current fragment terminates the I-VM:
- encrypts the current one;
- determines and decrypts the next fragment to be executed;
- updates the program counter of the Mon-VM.
System Block

- If $s$ is a system call or the first instruction of the program, a system block (SB) is the smallest program fragment that includes any instruction that may be executed in-between $s$ (not included) and either the next system call (included) or the program end;

- There is a distinct exit point for each system call that can be executed immediately after $s$, or for the program end if $s$ is the last call that may be executed before the end is reached.
System Block Graph

- A program may be described as a set of SBs and a system block graph (SBG) that denotes the execution order among these SBs.
- The SBG is an oriented graph where each node represents a distinct SB and each arc is coupled with a distinct system call among those executed by the program.
- In general, a node may be the source of several arcs, because there may be several exit points for each SB, each corresponding to the execution of a distinct system call.
Unit Block Graph

To map $\text{CFGraph}$ into the $SBG$, we consider an intermediate representation:

- a unit block (UB) is any sequence of instructions that belong to the same BB in-between two consecutive delimiters of the BB:
  - a delimiter is either a syscall or the first/last instruction of the BB.
- a unit block graph (UBG) is a transformation of $\text{CFGraph}$ in a graph that contains a node for each UB and an arc from the node representing UB$_1$ to the one representing UB$_2$ if:
  a) UB$_1$ and UB$_2$ belong to the same BB and UB$_2$ is executed immediately after UB$_1$, or
  b) UB$_1$ is the last UB of BB$_1$ and UB$_2$ is the first UB of BB$_2$ and BB$_2$ may be executed after BB$_1$. 
Algorithm to Map $CFGraph(P)$ into $SBG(P)$

1. splits each BB containing $n > 0$ system calls into $n + 1$ UBs;
2. generates $UBG(P)$;
3. visits $UBG(P)$ and, for each node $n$ that represents a UB:
   ▶ starts a depth first visit of the graph;
   ▶ determines $Succ(n)$, the set that includes any node $m$ that represents either a UB that may be executed after $n$ and that ends with a system call or the END block that terminates the program;
   ▶ merges all the UBs represented by nodes on the path from $n$ to any node in $Succ(n)$ into the same SB, $SB(n)$. 
Example: Control-Flow Graph
Example: Unit Block Graph
Example: System Block Graph
Run-Time Architecture of the Obfuscation Mechanism

1 the control-flow logic of the program to be obfuscated is partitioned between two VMs:
   ▶ the Mon-VM stores the obfuscated version of the SBs of $P$;
   ▶ the I-VM stores the information to deobfuscate $P$;

2 the I-VM encrypts and decrypts SBs at run-time in the memory of the Mon-VM so that only one SB is in clear.
Run-Time Architecture of the Obfuscation Mechanism (cont.)

3. To increase the complexity of an attack to rebuild the original code by accessing any of the executed SBs in clear, the Mon-VM only stores the SBs without any information about their execution order.

4. Transfer of control among these blocks is implemented by the I-VM by directly updating the program counter of the Mon-VM.

5. The new value of this register is computed through the SBG and a jump table that maps the SBs into their virtual address.
Run-Time Architecture
Virtual Interacting Network Community

- Virtual Interacting Network Community (Vinci) is a software architecture that exploits virtualization to share in a secure way an ICT infrastructure.
- Vinci decomposes users into communities: a set of users, their applications, a set of services and of shared resources.
- Users with distinct privileges and applications with distinct trust levels belong to distinct communities.
- Each community is supported by a virtual community network (VCN), i.e. a structured and highly parallel overlay that interconnects VMs built by instantiating one of a predefined set of VM templates.
VM Templates

A-VM = assurance VM
CC-VM = communication/control VM
FS-VM = file system VM
APP-VM = application VM
INF-VM = infrastructure VM
COM-VM = community VM

instantiation/configuration
join a community
insert the VM into a VCN

mapping

VMM

Node i

core 1 ... core n

VMM

Node j

... 
core 1 ... core m
A VINCI Node

- Infrastructure VM
  - manage node
  - configure VMs
  - setup policies

- File System VM
  - shared files
  - tainting module

- Community VM
  - community files
  - MAC/DAC module

- Assurance VM
  - VM introspection

- Application VM
  - client application
  - kernel

- Firewall VM
  - VPN/firewall module

- Virtual Machine Monitor
  - interconnection with other communities
  - interconnection with Application VMs of the same community in a distinct node
  - secure communication with a remote Application VM

LAN
Internet
Example: Communities and VCNs
Example: Tainting
Part IV

Final Remarks
Contributions of the Thesis (1)

- The definition of the process self in terms of a context-free grammar of system calls and invariants on the process state;
  - merge the ability of constraining the sequence of system calls with that of coupling memory assertions with such calls.
- PsycoTrace: a robust framework to check the process self in a fully transparent way:
  - no modifications either to the monitored process or the underlying OS so that the process is unaware of being monitored;
  - protect kernel integrity.
Contributions of the Thesis (2)

- A mechanism to bridge the semantic gap by transparently injecting an agent into the memory of a virtual machine.
- A framework that extends PsycoTrace to remotely attest the integrity of a node willing to join an overlay that generalizes the TPM that also considers the behavior of the node.
- The notion of system block and its adoption to both increase the accuracy and reduce the complexity of a static analysis to compute invariants of a program.
Contributions of the Thesis (3)

- An extension of PsycoTrace to protect a process against physical attacks by a novel code-obfuscation strategy that exploits virtualization to effectively split the program logic between two virtual machines.

- A strategy to manage and protect a cloud shared among users and applications with distinct trust and reliability levels:
  - the strategy exploits highly parallel overlays of virtual machines, where each virtual machine is an instance of a specialized template customized to run a small set of software components.
Ongoing Developments

The proposed measurements are security checks that are necessary but not sufficient. Currently, we are working on:

- definition of relation between accuracy and abstraction;
- formal results about the accuracy of the detection:
  - theorem for completeness: every attack will be detected?
  - theorem for soundness: every failed check means an attack?
References

Integrity and Cloud Security

Fabrizio Baiardi and Daniele Sgandurra

Fabrizio Baiardi and Daniele Sgandurra

[3] Protezione del Kernel Tramite Macchine Virtuali. Best Student Paper Award
Daniele Sgandurra and Fabio Campisi

Mihai Christodorescu, Reiner Sailer, Douglas Schales, Daniele Sgandurra, Diego Zamboni.

Mihai Christodorescu, Reiner Sailer, Douglas Schales, Arvind Seshadri, Daniele Sgandurra, Diego Zamboni.
In IBM AoT (Academy of Technology) Security and Privacy Symposium, June 22-26, 2009.
References

Francesco Tamberi, Dario Maggiari, Daniele Sgandurra, Fabrizio Baiardi
In proceedings of the Fourth IEEE International Conference on Information Assurance and Security (IAS 2008), Naples (Italy).

Daniele Sgandurra, Fabrizio Baiardi, Dario Maggiari and Francesco Tamberi

Fabrizio Baiardi, Dario Maggiari, Daniele Sgandurra and Francesco Tamberi
In Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP 2009), Weimar, Germany.
Vinci

Gaspare Sala and Daniele Sgandurra and Fabrizio Baiardi

Fabrizio Baiardi and Gaspare Sala and Daniele Sgandurra

Fabrizio Baiardi and Daniele Sgandurra

Fabrizio Baiardi and Daniele Sgandurra
In Proceedings of the 2nd International Conference on Autonomous Infrastructure, Management and Security Resilient Networks and Services (AIMS 2008), Bremen (Germany), LNCS 5127, pp. 65-78.

Fabrizio Baiardi, Dario Maggiari and Daniele Sgandurra
In Proceedings of HEALTHINF 2009 - International Conference on Health Informatics, Porto (Portugal).

[14] Securing a Community Cloud
Fabrizio Baiardi, Daniele Sgandurra.
VIMS and Other Topics

Fabrizio Baiardi, Diego Cilea, Daniele Sgandurra and Francesco Ceccarelli

Fabrizio Baiardi and Daniele Sgandurra and Claudio Telmon

Fabrizio Baiardi, Claudio Telmon and Daniele Sgandurra.

[18] Attestation of Integrity of Overlay Networks.
Fabrizio Baiardi and Daniele Sgandurra.
To appear in Journal of System Architecture, Special Issue on Security and Dependability Assurance of Software Architectures

Fabrizio Baiardi and Daniele Sgandurra.
Submitted to CCSW 2010: The ACM Cloud Computing Security Workshop