Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 236 (2009) 85–100
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

# Transparent Process Monitoring in a Virtual Environment

Fabrizio Baiardi[1] Dario Maggiari[2]

*Polo G. Marconi, Università di Pisa, La Spezia, Italy*

Daniele Sgandurra[3] Francesco Tamberi[4]

*Dipartimento di Informatica, Università di Pisa, Pisa, Italy*

**Abstract**

PsycoTrace is a system that integrates static and dynamic tools to protect a process from attacks that alter the process self as specified by the program source code. The static tools build a context-free grammar that describes the sequences of system calls the process may issue and a set of assertions on the process state, one for each invocation. The dynamic tools parse the call trace of the process to check that it belongs to the grammar language and evaluate the assertions. This paper describes the architecture of PsycoTrace, which exploits virtualization to introduce two virtual machines, the monitored and the monitoring virtual machines, to increase both the robustness and the transparency of the monitoring because the machine that implements all the checks is strongly separated from the monitored one. We discuss the modification to the kernel of the monitored machine to trace system call invocations, the definition of the legal traces and the checks to prove the trace is valid. We describe how PsycoTrace applies introspection to evaluate the assertions and analyze the state of the monitored machine and of its data structures. Finally, we present the security and performance results of the dynamic tools, and the implementation of the static tools.

*Keywords:* virtual machines, introspection, intrusion detection system

## 1 Introduction

Even if buffer overflows and format strings were discovered and analyzed in 1972 [2], they are still quite popular and effective nowadays. A classic attack that exploits a buffer overflow sends data to a network process coded in a type unsafe language such as C and, if the faulty listening process stores the data in an undersized stack buffer, the attack overwrites some information on the call stack. If the attacker

[1] Email: baiardi@di.unipi.it
[2] Email: maggiari@di.unipi.it
[3] Email: daniele@di.unipi.it
[4] Email: tamberi@di.unipi.it


doi:10.1016/j.entcs.2009.03.016

has properly crafted the data, she can overwrite the value of the function return pointer to implement a *control hijacking attack* that transfers control to malicious code in the data itself [1]. In 1988, the Morris worm exploited a buffer overflow. More recently, at least two Internet worms have exploited buffer overflows. In 2001, the "Code Red Worm" exploited a vulnerability in Microsoft's IIS, whereas in 2003 the "SQL Slammer Worm" damaged several hosts running Microsoft SQL Server.

PsycoTrace is a set of static and dynamic tools to protect a process program, i.e. the process *self*, from attacks that modify its intended behavior [28], such as those exemplified by buffer overflows. PsycoTrace static tools analyze the program source code and return both a context-free grammar that describes any sequence of system calls (i.e. *trace*) the process may generate, and a set of invariants on the process state, i.e. on the program variables. Each invariant is paired with a system call invocation and it constrains the values of the process variables and of the parameters of the system call. Starting from the data generated by the static tools, the dynamic tools verify that: (i) the trace currently generated by the process belongs to the language generated by the grammar; (ii) the assertion paired with the call is satisfied.

To check the run-time behavior in a robust and transparent way, PsycoTrace fully exploits virtualization by concurrently executing two virtual machines (VMs) on the same physical machine, namely the *monitored VM* and the *introspection VM*. The former executes the system that runs the monitored process, while the latter implements the monitoring system, i.e. it is a privileged VM that checks that the grammar is satisfied and exploits virtualization to directly access the physical memory of the monitored VM to inspect the process state and evaluate the assertions. Our approach is transparent because it does not require updating the code of the monitored process. However, the current version of PsycoTrace is not fully transparent because it inserts a module into the kernel of the monitored VM to intercept system calls and alert the introspection VM. The adoption of two VMs strongly increases the complexity of an attack against the dynamic tools, because an attacker should at first subvert in an undetected way the kernel of the monitored VM and then exploit a vulnerability of the virtual machine monitor to attack the introspection VM.

This paper presents PsycoTrace and it discusses the security results and the efficiency of the current implementation. Section 2 introduces the main concepts underlying the static tools that compute the expected behavior of a process and presents a first algorithm that is applied to the process source code to compute this behavior. Section 3 presents PsycoTrace's run-time architecture and the current prototype implementation. Section 4 shows the security evaluation of the prototype and a set of performance results. Section 5 reviews some related works. Finally, Section 6 draws a set of conclusions, discusses the current limitations and outlines future developments.

## 2 Static Tools and Process Expected Behavior

This section describes PsycoTrace static tools that compute the expected behavior of the process to be protected.

The main assumption underlying PsycoTrace is that a process can implement security critical operations through system calls only. Thus, provided that the kernel is trusted, an attempt to inject malicious code into a process memory can be detected by checking the sequence of system calls a process issues, i.e. the process trace, and the call parameters. These elements contribute to the definition of the process expected behavior and are computed by PsycoTrace static tools. The output of these tools is the input of the PsycoTrace dynamic tools that, as discussed in the next section, monitor the actual process behavior and compare it against the expected one.

As a first approximation of an optimal compromise between accuracy and efficiency, PsycoTrace static tools describe any trace the execution of $P$ may generate through a context-free grammar $CFG(P)$. At run-time, the trace generated by $P$ up to a given instant is legal or *coherent with CFG(P)*, if and only if it is a prefix of at least one string of $L(P)$, i.e. the language generated by $CFG(P)$. We assume that the high complexity of parsing due to a context-free grammar is justified by the better accuracy of the checks. As far as concerns the checks on the parameters of system calls, we generalize them by pairing an invariant $I(P, i)$ with each point $i$ of the program where $P$ issues a system call, i.e. $I(P, i)$ is an assertion on $P$ variables that holds any time $P$ reaches $i$. These checks increase PsycoTrace robustness with respect to *mimicry attacks* [29] that replace the parameters of a call without modifying the trace. For the time being, to verify the feasibility of our framework, we manually extract the assertions from our test programs. We plan to integrate the analysis that computes the invariants in the compilation phase, by properly analyzing the abstract syntax tree (AST) generated by the GCC compiler.

Terminal and non-terminal symbols of $CFG(P)$ depend upon the system calls that $P$ invokes and the functions defined in the source code. Formally, the context-free grammar $CFG(P)$ is a tuple $< T, F, S, R >$, where:

- $T$ is the set of terminal symbols, each corresponding to a distinct system call that $P$ invokes;
- $F$ is the set of non-terminal symbols, one for each function defined in the source;
- $S$ is the starting symbol, which corresponds to the `main` function;
- $R$ is the set of production rules.

We have defined an algorithm *GGA*, Grammar Generating Algorithm, to compute $CFG(P)$ by linearly analyzing each function defined in the source code of $P$. For each definition of a function `fun`, GGA adds a new symbol `FUN` to $F$ and a new rule $\mathtt{R}_{new}$ to $R$ with `FUN` as its left-hand-side. To generate the right-hand-sides of $\mathtt{R}_{new}$, GGA linearly analyzes the definition of `fun` and it applies the following rules to each block `B` of instructions inside `fun`:

- if `B` is a block of instructions without conditional or loop statements, GGA ap-

pends the system calls and functions invoked inside `B` in the same order to the right-hand-sides of $\mathtt{R}_{new}$;

- if `B` is a `if(cond)` {`block`$_1$} statement, GGA generates the rule:
  · `<A>` → `<B>` $|\epsilon$
  where `B` is a new non-terminal symbol that is the left-hand-side of the production generated by recursively applying GGA to `block`$_1$. Moreover, GGA appends the symbol `A` to the right-hand sides of $\mathtt{R}_{new}$;
- if `B` is a `if(cond)` {`block`$_1$} `else` {`block`$_2$} statement, GGA generates a new rule:
  · `<A>` → `<B>` | `<C>`
  where `B` and `C` are the left-hand-sides of the productions generated by the recursive application of GGA to `block`$_1$ and `block`$_2$ respectively. Furthermore, it appends the symbol `A` to the right-hand of $\mathtt{R}_{new}$;
- if `B` is a `if(cond)` {`block`$_1$} `else if(cond)` {`block`$_2$} ... `else if(cond)` {`block`$_n$} statement or, equivalently, a `switch` statement, it generates a new rule:
  · `<A>` → `<B`$_1$`>` | `<B`$_2$`>` | ... | `<B`$_n$`>`
  where `B`$_1$, `B`$_2$ ... `B`$_n$ are the left-hand-sides of the productions generated by the recursive application of GGA to `block`$_1$, `block`$_2$ ... `block`$_n$ respectively. Furthermore, it appends the symbol `A` to the right-hand of $\mathtt{R}_{new}$;
- finally, if `B` is a `while(cond)` {`block`$_1$} or, equivalently, a `for` cycle, GGA appends `(A)`$^*$ to the right-hand side of `R`, and it generates a new non-terminal symbol `A` that is the left-hand-side of the productions generated by the recursive invocation that applies GGA to `block`$_1$.

### *2.1 Meta-Compiler-Compiler Approach*

We currently build $CFG(P)$ by exploiting Bison [5] to generate a parser for the C language in which system calls are new tokens of the language. This parser implements GGA and its semantic actions generate $CFG(P)$. Moreover, we also exploit Bison to build a second parser that checks that the trace of $P$ is a string of $L(P)$. In more detail, we follow a meta-compiler-compiler approach (see Fig. 1) implemented in three steps. The first step does not depend upon the code of $P$ and each of the remaining steps builds a distinct parser:

(i) define an extended C grammar (ECG) in the Bison syntax in which system calls are added as new tokens. We also define the semantic actions of ECG that generate the system call grammar $CFG(P)$ by implementing GGA;

(ii) apply Bison to ECG to produce the parser that generates $CFG(P)$ from the source program of $P$. Future versions will include semantic actions for $CFG(P)$ that define the assertion that holds at each system call invocation;

(iii) apply Bison to $CFG(P)$ to build the parser that checks that $P$ generates a legal trace. The semantic actions paired with this parser include the evaluation of assertions.

Thus, in (i) a Flex-generated scanner recognizes system calls as tokens of the language. To create $CFG(P)$, the parsing of the source code in (ii) is decomposed into two further steps. In the first step, the semantic actions of the generated parser export an AST. The internal nodes of this tree represent either functions defined in the source code of $P$ or statements/expressions, whereas leaf nodes represent system calls. In the second step, the parser visits the AST to build $CFG(P)$ by applying GGA. $CFG(P)$ is represented through the Bison syntax so that in (iii) we can exploit Bison to generate the parser for $L(P)$ that is used at run-time to check that the trace of $P$ is legal.

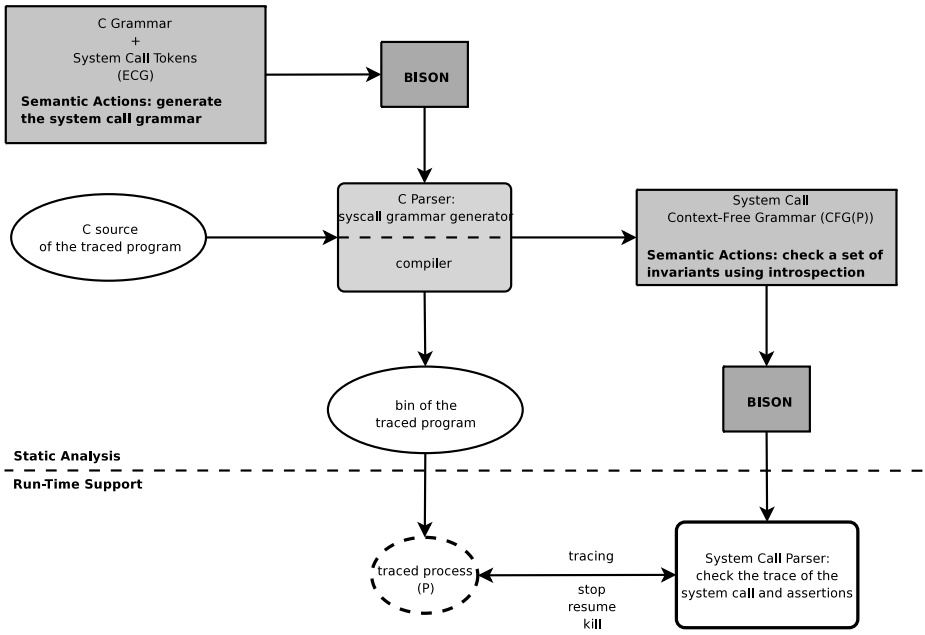


Fig. 1. Meta-Compiler-Compiler Approach.

## 3 Run-Time Architecture

PsycoTrace's run-time architecture monitors the actual process behavior and compares it against the expected behavior returned by the static tools and its implementation is built around virtualization and virtual machine introspection. The virtualization technology introduces the concept of *Virtual Machine Monitor* (VMM) [14], a thin software layer that creates, manages and monitors Virtual Machines (VMs), i.e. execution environments, which emulate the underlying physical machine. Virtualization simplifies the implementation of introspection, a generic technique that detect signs of intrusions by analyzing in detail the state of the machine to rebuild and check the consistency of data structures used either by the kernel or by user-level processes. In general, introspection requires specialized hardware units that access the physical memory of a machine [16]. However, by executing the monitored kernel and applications inside a VM, PsycoTrace can exploit *Virtual Machine Introspection* (VMI) [11] to implement introspection at the hardware/firmware level

without introducing additional units. VMI enables a privileged VM to analyze the state of the processes and kernel hosted on distinct VMs. It is very hard to elude or attack VMI, because it is implemented through a distinct VM and at a lower level than the one an attacker can access. Thus, the advantages of VMI are: (i) full visibility of the systems running on the VMs, because the VMM can access every VM component, such as the main memory or the processor's registers; (ii) more robustness, because the VMM is isolated from the monitored VM.

The input of dynamic tools defines the self of $P$ as computed by the static analysis and it consists of: (i) a context-free grammar $CFG(P)$ that defines the legal system call traces; (ii) a set of invariants $Inv(P) = \{I(P,1), \ldots, I(P,n)\}$, each paired with a program point $i$ where $P$ invokes a system call. The implementation of the run-time support introduces two virtual machines:

(i) the monitored VM (Mon-VM), i.e. the VM executing $P$;

(ii) 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, for example any memory region and any processor register, to inspect its running state and evaluate assertions on $P$ state, i.e. on the values of its variables. Each invariant is paired with a system call [23] and, in general, it constrains the values of the system call parameters or relates them to variable values. During the execution of $P$, the Mon-VM transfers control to the I-VM each time $P$ invokes a system call. At this point, the I-VM checks that the current trace satisfies $CFG(P)$ and it evaluates the invariant $I(P,i)$ paired with the point $i$ that $P$ has reached.

The important assumptions underlying the adopted approach are that: (i) the source code executed by $P$ is known; (ii) the VMM can be trusted; (ii) introspection safely extends the Trusted Computing Base (TCB). Two reasons support the latter two assumptions. Firstly, the VMM has full visibility of the Mon-VM, because it can access every components of it. Secondly, the VMM is more robust than commodity OSes because: (i) it exports a simple interface to the higher levels, which is more difficult to subvert than, for example, the one of a kernel that implements hundreds of system calls; (ii) the small size of the VMM code reduces the likelihood of a compromise and makes it possible to validate its correct implementation through a formal analysis. Notice that the kernel of the Mon-VM does not belong to the TCB because its integrity can be checked by the I-VM. In conclusion, since the VMM has full visibility of the VMs but it is strongly isolated from them, the complexity of compromising the VMM or of eluding the introspection monitoring capabilities of the Mon-VM is very high. Nonetheless, there are known threats against the VMM that also have to be considered [9].

Xen [6] is the adopted technology to create the VMs that implement the dynamic tools. We adopted the Xen VMM mainly for its high performance and complete integration with the Linux kernel. PsycoTrace exploits the para-virtualization approach, in which OSes are aware of the existence of the Xen VMM.

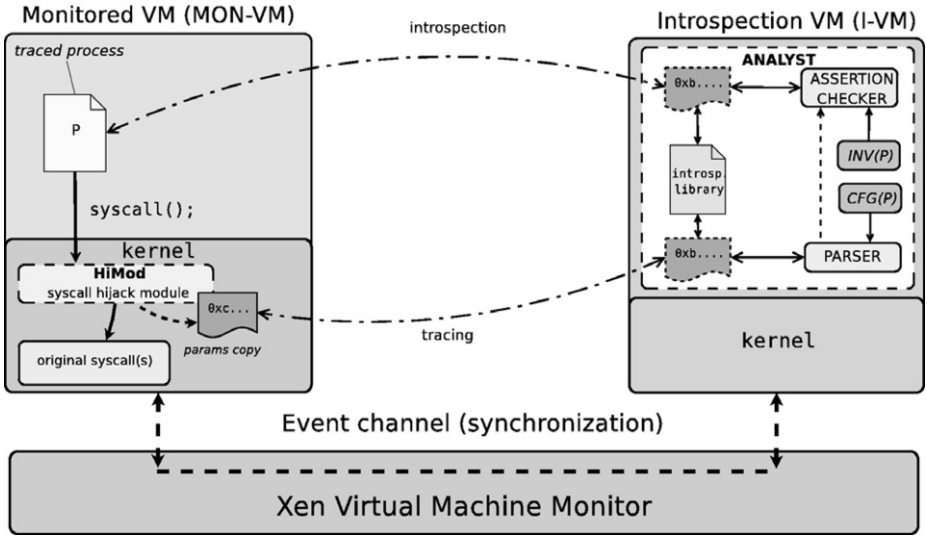


Fig. 2. Tracing a Process.

### *3.1 System Call Tracing*

The current dynamic tools are the *HiMod* and the *Analyst*, that run, respectively, in the kernel of the Mon-VM, and at the user-level of the I-VM. The HiMod is a kernel module that hijacks the system calls that $P$ invokes. Every time $P$ invokes a system call, HiMod notifies the Analyst, which can apply the consistency checks on the self of $P$. The Analyst includes the Bison-generated parser and an Assertion Checker that, respectively, check the traces of $P$ and evaluate the invariants. The Analyst starts by listening for communications from the HiMod about system call invocations. This interaction is synchronous, i.e. each time a system call is invoked, $P$ is paused and the execution is resumed only after the Analyst has completed the security checks. Notice that the ability of implementing the Analyst at the user level strongly simplifies its implementation with respect to the case in which it is implemented at the kernel level in the Mon-VM.

When HiMod is loaded, it allocates a kernel memory page to store the parameters of every system call that $P$ will invoke and the system call number. Then, the Analyst maps the shared page in its address space and, whenever a system call is invoked, it reads the system call number and the parameter values and then it resumes the execution of $P$ so that the current system call can be completed. To notify the Analyst that $P$ has issued a system call, HiMod allocates an *event channel* shared with the I-VM. This channel is a Xen data structure that emulates the interrupt mechanism. PsycoTrace uses this mechanism to synchronize the HiMod and the Analyst. When the Analyst allocates a new `evt_chn` data structure, it receives an integer value that represents the port number used to capture the notifications from the Mon-VM kernel module. At this point, the Analyst bounds itself to the specified port of the I-VM and waits a notification from the Mon-VM. Each notification corresponds to a system call invocation issued by $P$ (see Fig. 2).

PsycoTrace exploits the *XenStore* database, a common data-structure shared

by the VMs in the Xen architecture, which enables a privileged machine to access information about the running VMs. HiMod stores the base address of the shared page in a well-known token in the XenStore tree. The Analyst accesses the token and uses the address to map the page in its address space. The overall solution enables the Analyst to retrieve the information about the system call in the shared memory page.

Because of the large number of system calls in the Linux kernel, and since most of them are difficult to exploit to attack a process, PsycoTrace monitors only the system calls that are critical from the security point of view [3].

### *3.2 Virtual Machine Introspection*

Any time $P$ invokes a system call, the Mon-VM is suspended and the Analyst is informed. The Analyst knows the processor registers in the Mon-VM, the system call number, its parameters and it can retrieve the values of any program variable. As a consequence, the Analyst can check that both the resulting trace is legal and that the invariant paired with the call is verified. To check a trace is legal, HiMod transmits the system call number to the Analyst lexical analyzer which, in turn, returns the system call token to the parser as an input. If the current call does not belong to the terminal alphabet symbols, the parser returns an error. Otherwise, it checks the call by resuming the parsing from the point reached by the analysis of the previous call. Thus, PsycoTrace implements a *stream-oriented parsing* as opposed to the usual parsing of a whole sequence of tokens in a single step. This strongly simplifies the implementation with respect to the case where each invocation starts a new derivation from the starting symbol of the grammar and its parsing. The corresponding performance improvement may favor the adoption of a context-free grammar and of a GLR parser.

To evaluate the assertions in $Inv(P)$, we have implemented an *Introspection Library* which supports an x86/32 architecture, both regular paging and PAE. The library enables the Assertion Checker to access any memory region of $P$. The Introspection Library exploits the `libxc` library to access a page in the Mon-VM. The Introspection Library enables the Assertion Checker to traverse the page tables for $P$ to translate a virtual address of $P$ into a machine address, i.e. a physical address in Xen terminology. This address is used to map a page in the Assertion Checker process space. In this way, the Assertion Checker can map into its address space the pages of the traced process that store the values of interest to evaluate an assertion. The monitored process is killed as soon as PsycoTrace verifies that the trace is not coherent with the grammar, or that an assertion is false.

## 4 Effectiveness and Performance Tests

This section shows a preliminary evaluation of the attacks PsycoTrace can detect and a first set of results to evaluate the run-time overhead of the current implementation. The PsycoTrace dynamic tools are implemented through 450 lines of Perl code, which generate the HiMod by parsing the definition of the Linux system

calls. The HiMod consists of 2.5K lines of C code, while the Analyst is composed of about 3K lines of C code, including the Introspection Library. The definition of the extended C grammar including the semantic actions to generate $CFG(P)$ is about 1K lines of C++ code. Finally, the generated Bison parser for $CFG(P)$ is about 2.5K lines of C code.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <netdb.h>
#include <unistd.h>

#define BUFF 1024
#define SMALL_BUFF 512

int n = 10;

void logfile()
{
   int fd = open(``log.txt'', O_CREAT|O_WRONLY|O_APPEND, S_IRWXU);
   n--;
   write(fd, ``log'', 3);
   if(n>0) logfile();
   else (n=10);
   close(fd);
}

int parse_str(char *buff)
{
   char smallbuff[SMALL_BUFF];
   if(!strncmp(buff, ``copy'', 4)) strcpy(smallbuff, buff); /* VULNERABILITY*/
   if(!strncmp(buff, ``file'', 4)) logfile();
   if(!strncmp(buff, ``exit'', 4)) return 1;
   return 0;
}

int main()
{
   int fd, sockfd, ret, yes=1;
   socklen_t sin_size;
   struct sockaddr_in sin;
   struct hostent *h;
   char buffer[BUFF];
   fd = socket(AF_INET, SOCK_STREAM, 0);
   memset(&sin, 0, sizeof(sin));
   h = gethostbyname(``localhost'');
   sin.sin_family = AF_INET;
   sin.sin_port = htons(5555);
   sin.sin_addr.s_addr = INADDR_ANY;
   ret = setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
   ret = bind(fd, (struct sockaddr *) &sin, sizeof(sin));
   ret = listen(fd, 5);
   sin_size = sizeof(struct sockaddr_in);
   sockfd = accept(fd, (struct sockaddr *)&sin, &sin_size);
   dup2(sockfd, STDIN_FILENO);
   dup2(sockfd, STDOUT_FILENO);
   dup2(sockfd, STDERR_FILENO);
   while(1)
   {
      memset(buffer, 0, BUFF);
      read(sockfd, buffer, BUFF);
      if(parse_str(buffer) == 1) break;
      write(sockfd, ``ok\n'', 3);
   }
   close(sockfd);
}
```

Table 1
Testbed Server Program

```
<MAIN>: "dup2" "dup2" "dup2" ("read" <PARSE_STR>
        ("write")?)* "close".

<PARSE_STR>: (<LOGFILE>)?.

<LOGFILE>: "open" "write" (<LOGFILE>)? "close".
```

```
S0: DUP2 DUP2 DUP2 F0;
F0: READ F1 CLOSE |
    READ F1 WRITE F0 CLOSE;
F1: /* empty */ | F2;
F2: OPEN WRITE F2 CLOSE |
    OPEN WRITE CLOSE;
```

Table 2
Context-Free Grammar of The Example and its Bison Representation

### *4.1 Effectiveness*

First of all, we describe the process $P$ we used to test PsycoTrace effectiveness in attack detection. $P$ implements a single server application that opens a socket and reads from its stream a sequence of characters. Table 1 shows the source code of $P$. Table 2 shows the $CFG(P)$ generated by the GGA and the corresponding grammar in Bison syntax. Semantic actions are not shown. For the sake of conciseness, the $CFG(P)$ only describes the behavior of $P$ after the `accept` system call. Notice that the `LOGFILE` non-terminal generates a recursive production which defines the language `(open write)`$^n$`(close)`$^n$, which cannot be handled by a regular grammar. The `parse_str` function parses the received string. If the string begins with "copy", $P$ invokes `strcpy` to copy the receiving string in a local small buffer. `strcpy` is an insecure function that could be exploited to compromise the security of the application. If, instead, the received string begins with "file". $P$ invokes the `logfile` function. Lastly, if the string begins with "exit" the server closes the connection with the client. An example of a trace generated by the execution of $P$ is the following string: *(DUP2)*$^3$*; READ; (WRITE; READ; (OPEN; WRITE)*$^{10}$*; (CLOSE)*$^{10}$*)*$^2$*; (WRITE; READ)*$^2$*; CLOSE*.

We have implemented an attack that exploits the vulnerable `strcpy` function on the server-side, to manipulate a parameter from the client-side. The exploit overflows the server buffer by transmitting a string that contains a shellcode and that is built by appending to the string "copy" a sequence of `nop` instructions, the shellcode itself and finally a repetition of the jump address of the shellcode to overwrite the `parse_str` return address in the server stack. The execution of this exploit results in a remote shell with the privileges of the remote server process. The trace of $P$ after a successful attack is: *(DUP2)*$^3$*; READ;* ***SETUID;*** *BRK; OPEN; CLOSE; (OPEN; READ; CLOSE)*$^3$*; OPEN; CLOSE; (BRK)*$^3$*; TIME; BRK; IOCTL; BRK; (OPEN; READ; CLOSE)*$^2$*; BRK*. In this case, the PsycoTrace parser signals an inconsistency with respect to the expected behavior after the fourth system call, $P$ is stopped and no shell is spawned. The corresponding string is: `DUP2; DUP2; DUP2; READ; syntax error [SETUID]` → `process killed (pid=1054)`.

### *4.2 Performance Evaluation*

The system to run the prototype tools included a Pentium Centrino Duo T2250 1.7GHz. In all the tests, 128MB of physical RAM were allocated to the Mon-VM, running a Linux Debian distribution, and 874 MB RAM to the I-VM. The Xen

version was 3.1.0, while the Mon-VM Linux kernel version was 2.6.18-xen.

We evaluated the average time to execute the `bunzip2` tool to uncompress the Linux kernel on the Mon-VM in two cases, during a normal execution and when we only traced the `bunzip2` process. The execution time increased from 19.896 sec to 24.268 sec. The overhead, 21.97%, is rather high in this case because `bunzip` is a tool that invokes system calls at high rate. Table 3 shows the average execution time of three system calls executed on the Mon-VM in three cases: during the normal execution, while tracing the system calls and when the Analyst checks the trace and evaluates the assertions by accessing one page of $P$. In these tests, the traced program loops on each of such system call.

Fig. 3 displays the average execution time of the `time` system call in a loop when the Analyst evaluates the assertions, as the number of mapped pages varies from 1 to 10. Complex assertions correspond to a larger number of pages because they access several variables. The overhead is linear in the number of mapped pages.

Finally, we considered the execution time of the program described in Sect. 4.1, when a client generated and sent to $P$ a continuous stream of requests. Three cases were analyzed: (i) a normal execution of $P$; (ii) when system calls generated by $P$ are traced and notified to the Analyst but no check is applied; (iii) when tracing the system calls, checking the grammar and evaluating the assertions, by accessing one page of $P$. The total number of traced system calls generated by $P$ was 63234. In the worst case, a 48% overhead arises.

| system call | normal | traced | traced + introspection |
|---|---|---|---|
| time | 2 $\mu$sec | 55 $\mu$sec | 141 $\mu$sec |
| open | 3 $\mu$sec | 58 $\mu$sec | 116 $\mu$sec |
| write (1k buffer) | 8 $\mu$sec | 67 $\mu$sec | 177 $\mu$sec |

Table 3
Overhead of System Calls

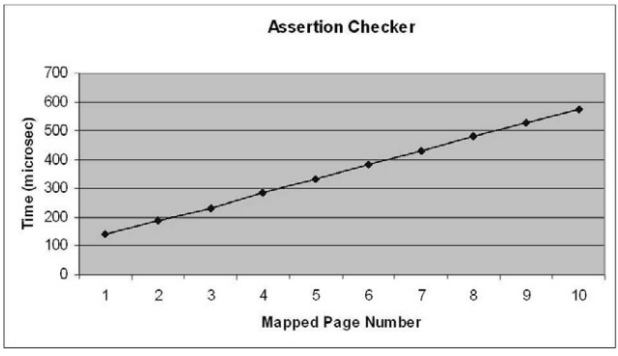


Fig. 3. Assertion Checker Overhead.

## 5 Related Work

Forrest et al. [10,15] firstly described a model that defines a process self through a set of short sequences of system calls. Any sequence that does not belong to the statistical-based set signals an intrusion. In [17,18] the authors propose a solution that pairs a program with a specification of its intended behavior, i.e. the program policy. The policy specification language is based upon predicate logic and regular expressions. A similar approach is discussed in [25], which exploits a language for capturing patterns of normal or abnormal behaviors of processes in terms of sequences of system calls and their arguments. An approach to detect anomalous program behaviors is proposed in [24], through a finite-state automaton (FSA) that learns a program behavior expressed as sequences of system calls, and it does not require access to the program source code. In [30,21] the authors propose data mining-based approaches to generate rules from system call sequences. Association rules and frequent episodes algorithms are used to compute the consistent patterns from audit data. Wagner and Dean [28] define a static analysis of the application source code that returns a specification of the expected application behavior. Intrusions are signaled by system call traces that are not coherent with the transition system that models the application. The paper introduces the *callgraph model*, built by analyzing the control-flow of the program. Then, this model is extended to the *abstract stack model* to take into account *impossible paths*. Finally, it considers a *digraph model* to simplify the implementation of the framework. Instead, PsycoTrace derives a context-free grammar directly from the source code rather than from the control-flow and the abstract stack. Furthermore, it exploits virtualization to apply introspection and take advantages of its isolation properties.

An approach to detect malicious system call through a static analysis of the binary program is proposed in [13], by building a model representing any remote call stream the process could generate. As the process executes remotely, the local agent operates on the model incrementally, ensuring that any call received does not violate the model. Each model is defined in terms of finite-state machines. Control-flow graphs generated from the binary code are used to build either a non-deterministic finite-state or a push-down automaton to mirror the execution control-flow of the executable. The *Dick Model* [12] includes a stack to record function call return locations, by using precalls and postcalls, and null system calls to eliminate impossible path and to simulate stack operations. *VtPath* [8] is an anomaly detection method that utilizes return addresses extracted from the call stack to generate the abstract execution path between two execution points in the program and it decides whether this path is valid according to what has been learned from the program normal runs. Moreover, since pushdown automata (PDA) model are rather inefficient because of non-determinism, Feng et al. [7] explore the *VPStatic* model, a variant of the VtPath model, which extracts context information about stack activity of the monitored program to define a deterministic model. *Paid* [19,20] is a compiler-based intrusion detection system that derives an accurate system call model from the application source code. It derives a deterministic FSA model that captures system call sites, their ordering and partial control-flow information. Moreover, Paid ex-

ploits run-time information to minimize the degree of non-determinism of the static analysis to extract the system call graph, and it also computes a set of constraints on the arguments of sensitive system calls. A kernel run-time verifier compares the process system call pattern against the statically derived model. An approach to prevent the execution of system calls due to code injection attacks is discussed in [22], which exploits a table of addresses of allowed system calls to decide whether a system call was executed inside the injected code.

*Janus,goldberg96secure* is a sandbox-based secure environment that restricts an untrusted application access to the OS by tracing the process activities. Similarly, *Ostia,ostia* is a system call interposition-based sandbox system that relies on a delegating architecture, which includes a kernel module to enforce a hard-coded policy to prevent the execution of sensitive calls, and a user-level application that accesses sensitive resources on behalf of the sandboxed process. When the process issues a sensitive system call, an emulation library intercepts the call and transmits the request to the user-level agent via an IPC channel. Among the tools to protect a process, *StackGuard,cowanstackguard* is a kernel patch aimed at preventing smash stacking attacks by protecting the return address on the stack. StackGuard places a *canary* word, i.e. a randomly generated value, next to the return address and it generates code to check this value each time a function is called. If the canary has been altered when the function returns, then some attempt to overflow the stack has been successful. Moreover, *libsafe* library [26] provides a way to invoke insecure functions in a secure way.

## 6 Current Status, Limitations and Developments

PsycoTrace integrates static tools, which build a specification of the process expected behavior, with dynamic tools that exploit virtualization technology to build a robust monitor of the actual process behavior. If the dynamic tools detect either a system call that is incompatible with the grammar, or a false assertion, PsycoTrace deduces that the process has been successfully attacked and it kills it. No false positive is possible, because the specification over-approximates the behavior of the program. Furthermore, this specification strongly reduces the number of false negatives due to conditional statements, unbounded iterations and recursion. We believe that PsycoTrace can prevent and detect most attacks against the program self, i.e. attacks that modify its intended behavior. The current status of the work shows promising results and an acceptable performance overhead.

The model underlying the definition of PsycoTrace is focused on those attacks that modify the behavior of a process as expressed in the source code, such as those that inject and execute malicious code. As a consequence, there are several security problems that PsycoTrace does not cover which are due to flawed application logic, such as time of check time of use (TOCTOU) errors [4]. Moreover, there are several non-standard control-flows that a static analysis cannot handle very easily. As an example, a function pointer could be used to indirectly invoke a system call. To take into account function pointers, the static analysis should predict all the possible

targets of every indirect call through a function pointer. For the time being, we neglect function pointers, so our approach may miss some system call invocation. Moreover, we currently do not handle `GOTO` statements and inline assembly.

Additionally, we neglect linked libraries, such as the Gnu C library (`glibc`). There are two approaches to deal with this problem. If the source code of the library is available, we can apply the static analysis previously discussed. If, instead, the source code is not available, we can apply some reverse-engineering technique, such as disassembly [27], to define the context-free grammar paired with each library function. Since our approach exploits Bison to generate the system-call context-free grammar, this would require the definition of an extended assembly grammar, instead of the extended C grammar, in the first step of the meta-generation approach. The handling of dynamic linking is even more complex, because a process can load at run-time any library by exploiting a function such as `dlopen()`. This requires a run-time update of both the grammar of system call invocations and of the set of assertions, each time a new library is loaded. In both cases, an alternative solution inserts *null calls* into the program code to signal the invocation of a library function to stop/resume the consistency checks.

A further aspect of interest is related to system call invocations. In fact, the source code can directly invoke a system call without using the C library wrapper functions. Thus, direct invocations of system calls, using the `int $0x80` or `sysenter` assembly instructions should also be located and correctly decoded. Finally, another non-standard control-flow mechanism that, for the time being, we neglect, is the OS signal facility, which enables a process to register a callback function so that each time it receives a particular signal it can handle it with the corresponding callback function. This handler could issue one or more system calls as well and, since signals may be delivered at any moment, a static analysis cannot deduce their order. Thus, it would be cumbersome to consider all the possible ordering of OS signals because we should take into account that a process $P$ may receive a signal after each system call, and the corresponding handler may invoke a sequence of system calls.

A last issue to be considered is monitoring transparency. In the current version, the monitoring is fully transparent with respect to the application because it is not modified. Instead, transparency is not achieved with respect to the OS of the Mon-VM because of the tracing module that it is inserted into the kernel. We are currently developing a fully transparent solution based upon interception of `int` instructions at the hardware/firmware level, by exploiting recent processors' features.

In the current PsycoTrace prototype, the HiMod monitors a single PID. This implies that a process that forks some children processes is not correctly monitored. Thus, PsycoTrace also needs to monitor the children, for example by retrieving their PID through introspection anytime a `fork` is called. Finally, we are currently working on the definition of the static tools that extract the set of assertions from the source code.

# References


[1] Aleph One, *Smashing the stack for fun and profit*, Phrack **7** (1996).

[2] Anderson, J. et al., *Computer Security Technology Planning Study*, Technical report, ESD-TR-73-51 (1972).

[3] Bernaschi, M., E. Gabrielli and L. V. Mancini, *Operating system enhancements to prevent the misuse of system calls*, in: *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security* (2000), pp. 174–183.

[4] Bishop, M. and M. Dilger, *Checking for Race Conditions in File Accesses*, Computing Systems **2** (1996), pp. 131–152.

[5] Donnelly, C. and R. Stallman, "The Bison manual: using the YACC-compatible parser generator," Boston, MA: GNU, 2006.

[6] Dragovic, B., K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham and R. Neugebauer, *Xen and the Art of Virtualization*, in: *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[7] Feng, H. H., J. T. Giffin, Y. Huang, S. Jha, W. Lee and B. P. Miller, *Formalizing Sensitivity in Static Analysis for Intrusion Detection*, in: *IEEE Symposium on Security and Privacy*, Oakland, California, 2004.

[8] Feng, H. H., O. M. Kolesnikov, P. Fogla, W. Lee and W. Gong, *Anomaly Detection Using Call Stack Information*, in: *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy* (2003), p. 62.

[9] Ferrie, P., *Attacks on Virtual Machine Emulators*, Symantec Security Response, December (2006).

[10] Forrest, S., S. A. Hofmeyr, A. Somayaji and T. A. Longstaff, *A Sense of Self for Unix Processes*, in: *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy* (1996), pp. 120–128.

[11] Garfinkel, T. and M. Rosenblum, *A Virtual Machine Introspection Based Architecture for Intrusion Detection*, in: *Proc. Network and Distributed Systems Security Symposium*, 2003.

[12] Giffin, J., S. Jha and B. Miller, *Efficient context-sensitive intrusion detection*, in: *11th Annual Network and Distributed Systems Security Symposium (NDSS), San Diego, California*, 2004.

[13] Giffin, J. T., S. Jha and B. P. Miller, *Detecting Manipulated Remote Call Streams*, in: *Proceedings of the 11th USENIX Security Symposium* (2002), pp. 61–79.

[14] Goldberg, R. P., *Survey of virtual machine research*, IEEE Computer **7** (1974), pp. 34–45.

[15] Hofmeyr, S. A., S. Forrest and A. Somayaji, *Intrusion Detection Using Sequences of System Calls*, Journal of Computer Security **6** (1998), pp. 151–180.

[16] Jr., N. L. P., T. Fraser, J. Molina and W. A. Arbaugh, *Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor*, in: *USENIX Security Symposium*, 2004, pp. 179–194.

[17] Ko, C., G. Fink and K. Levitt, *Automated detection of vulnerabilities in privileged programs by execution monitoring*, in: *Proceedings of the 10th Annual Computer Security Applications Conference* (1994), pp. 134–144.

[18] Ko, C., M. Ruschitzka and K. Levitt, *Execution monitoring of security-critical programs in distributed systems: a specification-based approach*, in: *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy* (1997), p. 175.

[19] Lam, L.-C. and T.-C. Chiueh, *Automatic Extraction of Accurate Application-Specific Sandboxing Policy*, in: *RAID*, 2004, pp. 1–20.

[20] Lam, L.-C., W. Li and T.-C. Chiueh, *Accurate and Automated System Call Policy-Based Intrusion Prevention*, in: *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)* (2006), pp. 413–424.

[21] Lee, W. and S. Stolfo, *Data mining approaches for intrusion detection*, in: *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.

[22] Linn, C. M., M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray and J. H. Hartman, *Protecting against unexpected system calls*, in: *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium* (2005), pp. 16–16.

[23] Saïdi, H., *Guarded models for intrusion detection*, in: *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security* (2007), pp. 85–94.

[24] Sekar, R., M. Bendre, D. Dhurjati and P. Bollineni, *A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors*, in: *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy* (2001), p. 144.

[25] Sekar, R. and P. Uppuluri, *Synthesizing fast intrusion prevention/detection systems from high-level specifications*, in: *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium* (1999), pp. 6–6.

[26] Tsai, T. K. and N. Singh, *Libsafe: Transparent System-wide Protection Against Buffer Overflow Attacks*, in: *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks* (2002), p. 541.

[27] Vigna, G., "Malware Detection," Advances in Information Security, Springer, 2007 .

[28] Wagner, D. and D. Dean, *Intrusion Detection via Static Analysis*, in: *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy* (2001), p. 156.

[29] Wagner, D. and P. Soto, *Mimicry attacks on host-based intrusion detection systems*, in: *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security* (2002), pp. 255–264.

[30] W.Lee, S. and P.K.Chan, *Learning patterns from UNIX processes execution traces for intrusion detection*, AAAI Workshop on AI Approaches to Fraud Detection and Risk Management, 1997, pp. 50–56, aAAI Press.