# A Smell of Orchids

Jean Goubault-Larrecq[1] and Julien Olivain[1,2]

[1] LSV, ENS Cachan, CNRS, INRIA
LSV, 61 avenue du président Wilson, F-94235 Cachan Cedex
{olivain,goubault}@lsv.ens-cachan.fr
[2] Above Security, Suite 203
1919 Lionel-Bertrand boulevard, Boisbriand, Québec, Canada, J7H 1N8
julien.olivain@abovesecurity.com

**Abstract.** Orchids is an intrusion detection tool based on techniques for fast, on-line model-checking. Orchids detects complex, correlated strands of events with very low overhead in practice, although its detection algorithm has worst-case exponential time complexity.

The purpose of this paper is twofold. First, we explain the salient features of the basic model-checking algorithm in an intuitive way, as a form of dynamically-spawned monitors. One distinctive feature of the Orchids algorithm is that fresh monitors need to be spawned at a possibly alarming rate.

The second goal of this paper is therefore to explain how we tame the complexity of the procedure, using abstract interpretation techniques to safely kill useless monitors. This includes monitors which will provably detect nothing, but also monitors that are subsumed by others, in the sense that they will definitely fail the so-called shortest run criterion. We take the opportunity to show how the Orchids algorithm maintains its monitors sorted in such a way that the subsumption operation is effected with no overhead, and we correct a small, but definitely annoying bug in its core algorithm, as it was published in 2001.

## 1 Introduction

It is a *lieu commun* that the security of computer systems and networks is more and more challenged by new threats. Viruses, worms, Trojan horses have been reported to infect computers since the early 1980s, network attacks such as denial of service, spoofing, defacing have been commonplace since the late 1980s, and new attacks keep coming up, either based on new principles such as phishing or keyloggers, or using older vulnerabilities. New applications create new opportunities for vulnerabilities. E.g., the advent of Web-based applications created new families of vulnerabilities such as SQL insertion, PHP insertion, or cross-site scripting.

It is harder and harder to maintain an acceptable level of security on computers and networks, while keeping the induced nuisance at an acceptable level to honest users. Static analysis and formal methods in general can certainly help increase the faith we can put in critical pieces of code, but they are far from being able to ascertain the global security of a whole computer system or network.

M. Leucker (Ed.): RV 2008, LNCS 5289, pp. 1–20, 2008.

A successful family of techniques in this respect is *intrusion detection*, whereby flows of system and network events are monitored in real time, and analyzed so as to detect attacks. Intrusion detection systems that also react against attacks are sometimes called intrusion *prevention* systems.

Definitions in this domain tend to be fuzzy, starting from the very notion of attack. *Anomaly detection* systems count as possible attack any significant statistical deviation from normal behavior. *Misuse detection* systems would check the flow of events against some security policy, raising an alert when the policy is violated, or against some database of attack signatures, raising an alert when one of the signatures is matched.

ORCHIDS [6] is an intrusion prevention system that was developed at LSV by the authors, starting from 2002. It was initially meant as a misuse detection system, whose originality was that it could detect complex attacks consisting of several events correlated over time. An example of such an attack is the `ptrace` attack [10,11], which we shall describe shortly in Section 2. We shall again use this attack to describe the ORCHIDS detection algorithm by means of an example run, in Section 3. In Section 4, we shall describe the core detection algorithm in more detail, repairing a bug in [13]. The point of this algorithm is to detect the shortest run by keeping all runs sorted with the lowest possible overhead—in particular, we *never* call any sort routine. The `ptrace` example, while impressive, remains simple-minded, for reasons we shall explain in Section 6. There, we shall illustrate how a single signature can detect whole families of attacks, and even some zero-day attacks. This is important to security practitioners.

ORCHIDS was presented at the CAV'05 conference [7], and its core algorithm is based on the one described in [13, Section 4]. In these papers, ORCHIDS was described as a model-checker for a specific temporal logic. However, somehow ORCHIDS is better described as running monitors, with the twist that each monitor will spawn new monitors dynamically, to follow possible beginnings of attacks. Presenting this work at RV'08 is therefore quite apt indeed, and we must thank Martin Leucker and the organizers for inviting the first author to Budapest and allow him to give an overview of it.

## 2 The `ptrace` Attack Example

Let's concentrate on the `ptrace` attack [10,11]. This is a local-to-root exploit, i.e., it enables a user having local access to a host machine to get root privileges. This is a real attack, which has been used in practice. Patches have been available for some time, of course; none of the attacks presented here should be effective on up-to-date systems.

The main point in using the `ptrace` attack as an example is that it is witnessed by a flow of events that are all entirely uncharacteristic of any malicious activity in isolation: most events in an instance of the attack are calls to the `ptrace` system call, a perfectly benign system call used for all debugger-related activities. Rather, the *sequence* of events throughout the attack must be identified to isolate

the attack. In other words, the `ptrace` attacks avoids detection by classical intrusion detection systems, which only match individual events against a database of word patterns.

To understand the attack, it is useful to realize what a modular operating system kernel, such as most versions of Linux, will do when a user program calls an unimplemented kernel functionality. See Figure 1, where the user program has pid 100, and the unimplemented functionality is the special case of the `socket` system call on the (never implemented, Linux specific) domain `AF_SECURITY`. The kernel will search for a kernel module implementing this, calling the `modprobe` utility to search and install the desired module. If the search fails, an error code is reported.

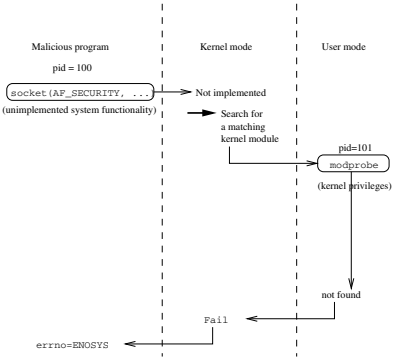
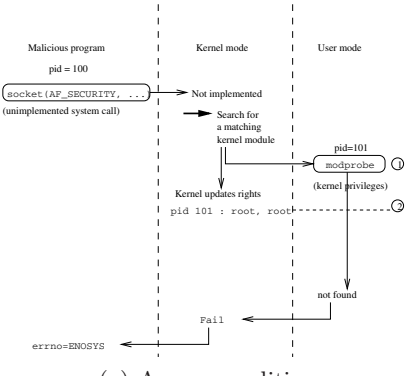
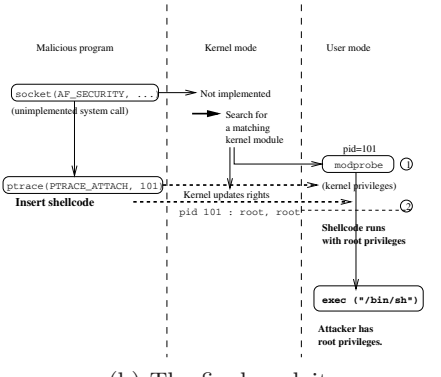



**Fig. 1.** Calling an unimplemented kernel functionality

While this is how this is meant to work, some versions of Linux suffer from a race condition (Figure 2(a)). While `modprobe` has started running, with kernel privileges, the kernel updates the owner tables to make `modprobe` root-owned instead of user-owned. So there is a small amount of time where the malicious user program has complete control over the kernel process `modprobe`: between timepoints ① and ②. The malicious program takes this opportunity to attach the `modprobe` process through the standard Unix debugging API function `ptrace`, and to insert a *shellcode* (a code of the intruder's choosing) inside it. When `modprobe` resumes execution, it will execute the shellcode with full root privileges (Figure 2(b)).




(a) A race condition

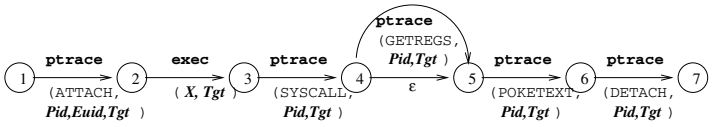



(b) The final exploit

**Fig. 2.** The `ptrace` Linux attack

## 3 Detecting the ptrace Attack

ORCHIDS can be made to detect this attack using the following signature:



(1)

This can be thought as an automaton (a *monitor*), with starting state ①, and final state ⑦. Transitions are labeled with *patterns*, say ptrace(ATTACH, $Pid$, $Euid$, $Tgt$), meant to match single events such as ptrace(ATTACH, 57, 500, 58) (with the variable $Pid$ mapped to 57, $Euid$ to 500, $Tgt$ to 58; ORCHIDS actually uses explicit field selectors instead of patterns—we use patterns to simplify the exposition). Transitions can also be labeled with the symbol $\epsilon$: these can be triggered without matching any event.

Patterns may contain variables, i.e., signatures have first-order capabilities. However, the main difference with standard monitors is that such an automaton is meant to match *subsequences* of the event flow, not the whole sequence of events. For example, the ptrace signature above should match the subsequence of the event flow shown in Figure 3 (see Section 3.2) consisting of events number 3, 4, 7, 8, 9, 12 with $Pid = 100$, $Euid = 500$, $Tgt = 101$. It should also match the sequence of events 3, 4, 7, 9, 12, omitting event 8 by choosing to go through the $\epsilon$ transition between states ④ and ⑤ instead of going through the transition labeled ptrace(GETREGS, $Pid, Tgt$). Note that it should also match the sequence of events 3, 4, 7, 8 (optional), 10, and 12, and also the sequence 3, 4, 7, 8 (optional), 11, and 12.

To fix ideas, let *events* be ground first-order terms over some set of function symbols (e.g., ptrace, exec). This signature includes numbers such as 100, 101, or 58 as constants, as well as symbolic values and character strings such as GETREGS. (Interpreting actual events, such as provided by the Linux kernel module Snare or other input modules, as terms, is essentially a parsing task.) *Patterns* are just first-order terms, not necessarily ground. We take the set $\mathcal{V}$ of variables to be the disjoint union of two countably infinite subsets, the set $\mathcal{V}_r$ of so-called *rigid* variables and $\mathcal{V}_f$ of *flexible* variables. Rigid variables such as $Pid$, $Euid$, or $Tgt$ above are meant to match the same value over all events in a matching subsequence, while flexible variables may assume distinct values at each event. This is reminiscent of Manna and Pnueli [2]. ORCHIDS actually imposes a typing discipline on variables, events, and patterns, of which the distinction between rigid and flexible is just one aspect. We shall largely ignore the details of this typing discipline, except in Section 5.

Let $\mathcal{T}(\mathcal{V})$ be the set of all terms (patterns), $\mathcal{T}$ be the subset of all ground terms (events). We let $\mathrm{fv}(t)$ denote the set of free variables in $t$, $t\sigma$ denote the result of applying the substitution $\sigma$ to $t$, where substitutions $\sigma$ are finite maps $[x_1 := t_1, \ldots, x_n := t_n]$ with $x_1, \ldots, x_n$ pairwise disjoint variables—in which case the *domain* $\mathrm{dom}\,\sigma$ of $\sigma$ is $\{x_1, \ldots, x_n\}$. Substitutions $\sigma$ are meant to keep

```
1: open ("/etc/passwd", "r", 58, 500)
2: ptrace (ATTACH, 57, 500, 58)
3: ptrace (ATTACH, 100, 500, 101)
4: exec ("modprobe", 101)
5: ptrace (ATTACH, 100, 500, 101)
6: exit (58)
7:  ptrace (SYSCALL, 100, 101)
8:  ptrace (GETREGS, 100, 101)
9:  ptrace (POKETEXT, 100, 101)
10: ptrace (POKETEXT, 100, 101)
11: ptrace (POKETEXT, 100, 101)
12: ptrace (DETACH, 100, 101)
```

**Fig. 3.** A typical event flow

the values of specific variables such as $Pid$ or $Euid$ above. Let $\sigma \oplus \sigma'$ be the substitution with domain $\mathrm{dom}\,\sigma \cup \mathrm{dom}\,\sigma'$, mapping every $x \in \mathrm{dom}\,\sigma'$ to $\sigma'(x)$, and every $x \in \mathrm{dom}\,\sigma \setminus \mathrm{dom}\,\sigma'$ to $\sigma(x)$.

Given a substitution $\sigma$, a pattern $p$ and a ground term $t$, we let $\sigma \vdash p \lhd t \Rightarrow \sigma \oplus \sigma'$, provided the most general matcher $\sigma'$ of $p$ against $t$ exists, and $\sigma(x) = \sigma'(x)$ for every $x \in \mathcal{V}_r \cap \mathrm{dom}\ \sigma \cap \mathrm{dom}\ \sigma'$ (i.e., we check that rigid variables do not change; flexible variables may be overwritten at will). In this case, we say that pattern $p$ *matches* event $t$ in $\sigma$, yielding $\sigma \oplus \sigma'$. E.g., $\mathtt{ptrace}(\mathtt{ATTACH}, Pid, Euid, Tgt)$ matches $\mathtt{ptrace}(\mathtt{ATTACH}, 57, 500, 58)$ (event number 2 in Figure 3) in the empty substitution, yielding $[Pid := 57, Euid := 500, Tgt := 58]$; $\mathtt{ptrace}(\mathtt{SYSCALL}, Pid, Tgt)$ matches $\mathtt{ptrace}(\mathtt{SYSCALL}, 100, 101)$ (event 7) in $[Pid := 100, Euid := 500, Tgt := 101]$ but not in $[Pid := 57, Euid := 500, Tgt := 58]$.

Each transition in a signature may be additionally labeled with a *guard*, which is an expression over the variables in $\mathcal{V}$ denoting a Boolean value. The actual syntax of guards is unimportant here. Letting $\mathcal{G}$ be the set of guards, we shall only assume that one may compute the finite set $\mathrm{fv}(g)$ of free variables in the guard $g$, and that we may evaluate a guard $g$ in an environment $\sigma$ to a Boolean value $[\![g]\!]\sigma$, as soon as $\mathrm{fv}(g) \subseteq \mathrm{dom}\,\sigma$.

*Signatures* $\Sigma$ are automata $(Q, I, T, \Delta)$, where $Q$ is a finite set of *states*, $I \subseteq Q$ is the subset of *initial states*, $T \subseteq Q$ is the set of *final states*, and $\Delta \subseteq Q \times (\mathcal{T}(\mathcal{V}) \uplus \{\epsilon\}) \times \mathcal{G} \times Q$ is the *transition relation*. Any transition of the form $(q_0, \epsilon, g, q_1)$ is called an $\epsilon$-*transition*. We assume that no $\epsilon$-transition goes out of the initial state, i.e., that there is no transition of the form $(q_0, \epsilon, g, q_1)$ with $q_0 \in I$.

An *event flow* $t_\bullet$ is any finite or infinite sequence $t_1 t_2 \ldots t_i \ldots$ of events, i.e., of ground terms in $\mathcal{T}$. We are interested in finding specific subsequences of events with indices $i_1 < i_2 < \ldots < i_k$ $(k \geq 1)$: these subsequences are uniquely determined by the sets $\{i_1, i_2, \ldots, i_k\}$, which we call *subflows*. A *partial run* of an event flow $t_\bullet$ against a signature $\Sigma = (Q, I, T, \Delta)$ is a sequence $q_0, \sigma_0 \xrightarrow{i_1} q_1, \sigma_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} q_k, \sigma_k$, where $k \geq 1$, $q_0, q_1 \ldots, q_k$ are states in $Q$, $q_0 \in I$, $\sigma_0$ is the empty substitution, and there is an integer $i_{k+1}$ such that for all $j$, $1 \leq j \leq k$, either there is a transition $(q_{j-1}, \epsilon, g, q_j) \in \Delta$ with $[\![g]\!]\sigma_{j-1}$ true and $i_j = i_{j+1}$ (go through the $\epsilon$-transition, do not move in the event flow), or there is a transition $(q_{j-1}, p, g, q_j) \in \Delta$ with $p \neq \epsilon$, $\sigma_{j-1} \vdash p \lhd t_{i_j} \Rightarrow \sigma_j$, with $[\![g]\!]\sigma_j$ true, and $i_j < i_{j+1}$ (go through the transition, acquiring new values for variables, and proceed to some later point in the event flow). The subflow of such a partial run

is the set of indices $i_1$, $i_2$, ..., $i_k$, with duplicates removed. We say that $i_1$ is its *birthdate*. A *complete run* is a partial run such that, additionally, $q_k \in F$.

ORCHIDS is in fact based on a more complex, and more expressive, language of signatures, with mutable variables, external system calls, and an embedded Prolog interpreter to maintain various databases: black lists, attacks that have succeeded in the past and that may be prerequisites to some others, neighboring relations between hosts in networks, equivalences between host names and between other services, and alert correlation information as in the M2D2 model [5]. However, the above simpler automata are enough to convey the essential ideas.

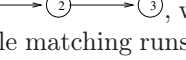
### 3.1 Shortest Runs

It is important to note that there is no unique complete run of a given event flow against a given signature in general, as we have seen above on the example of the `ptrace` attack: even the corresponding subflows are not unique.

An intrusion detection system cannot just report the *existence* of a matching subsequence (an attack): it should also collect, report enough information about the attack, and use it to react appropriately. Complete runs are enough information. On the other hand, it cannot report *all* matching complete runs either. This would flood the security administrator with too many alerts, prompting him to turn the intrusion detection system off, or to ignore its warnings. Instead, ORCHIDS reports a *shortest run* [13] among all matching subsequences starting at a given event. The definition is as follows. For any subflows $i_1 < i_2 < \ldots < i_k$ and $j_1 < j_2 < \ldots < j_\ell$ $(k, \ell \geq 1)$, we let $(i_1, i_2, \ldots, i_k) \preceq (j_1, j_2, \ldots, j_\ell)$ iff $i_1 = j_1$ (the subflows have the same birthdate), $i_k \leq j_\ell$ (the first one stops earlier than the second one), and $(i_1, i_2, \ldots, i_k)$ is lexicographically smaller than $(j_1, j_2, \ldots, j_\ell)$.

On subflows with a given, fixed birthdate $i_1$, $\preceq$ is a total well-founded ordering, so any non-empty family $F$ of subflows with the same birthdate $i_1$ has a unique smallest element wrt. $\preceq$. This is the *shortest subflow* of $F$. By extension, a *shortest run* of a flow $t_\bullet$ against a signature $\Sigma$ with birthdate $i_1$, is a complete run whose subflow is shortest, among all subflows of complete runs against $\Sigma$ with birthdate $i_1$.

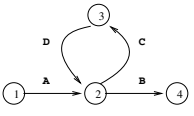
ORCHIDS will only return shortest runs, taken as canonical representatives of all runs against a given signature $\Sigma$ with a given birthdate $i_1$. Another view is to say that ORCHIDS considers all runs against the same signature $\Sigma$ and starting at the same position as equivalent. Pouzol and Ducassé [9] consider more general notions of equivalence. However, the efficiency of the algorithm of Section 4 owes a lot to our particular definition of equivalence. While the latter is fixed in ORCHIDS, experience shows that it is adequate. It was argued in [13] that the shortest run against a given signature with birthdate $i_1$, was in a sense the most informative one, and experience again has vindicated this stance. We discuss this briefly.

First, shortest runs are shortest in the intuitive sense that they can be reported as soon as one run succeeds that matches the given signature. A simple example is the signature 

, with some arbitrary event `A`, and the event flow `AAAAA...AA`. While matching runs with birthdate $i_1 = 1$ include all pairs $1, n$ for

all $n \geq 2$, only the pair $1, 2$ counts as shortest. This guarantees that the intrusion detection system will react as soon as some matching run is encountered.

Second, and more subtly, consider the signature shown on the right, and the event flow ACDCDCDCB. Any shortest run with birthdate $i_1 = 1$ must end at $i_k = 9$, on the final B. Candidates are $1, 9$, which only matches the initial A and the final B; or $1, 2, 3, 9$, which additionally matches the first CD, going around the loop between states ② and ③; or $1, 4, 5, 9$; or $1, 4, 7, 9\ldots$ we invite the reader to check that the shortest run is $1, 2, 3, 4, 5, 6, 7, 9$: contrarily to what the adjective "shortest" may suggest, the shortest run contains as many relevant events as permitted to describe a matching attack.




Returning to the ptrace attack signature (1), and the example event flow of Figure 3, the only matching runs have birthdate $i_1 = 3$, and the only shortest run is $3, 4, 7, 8, 10, 12$. Note that the optional event 8 is included, although it would be allowed to skip it, by going through the $\epsilon$-transition from ④ to ⑤, instead of that labeled ptrace(GETREGS, $Pid, Tgt$). The latter transition would be irrelevant without the shortest run semantics. Here, it instructs Orchids to report an event of the form ptrace(GETREGS, $Pid, Tgt$) in a matching attack, in case one is indeed present.

### 3.2 Running Orchids on the ptrace Signature

Let us simulate an execution of Orchids of the signature (1) against the example event flow of Figure 3. This will give us an opportunity to illustrate the salient features of the Orchids algorithm, which we shall explain in more detail in Section 4. Here Orchids will try to match just one signature; in normal use, it will try to match all signatures in a given signature database at the same time.

Initially, Orchids reads event 1. Since (1) does not contain any pattern matching an open event, we skip to event 2, $t =$ ptrace(ATTACH, 57, 500, 58). The pattern $p =$ ptrace(ATTACH, $Pid, Euid, Tgt$) matches this, i.e.: $\sigma_0 \vdash p \lhd t \Rightarrow [Pid := 57, Euid := 500, Tgt := 58]$. So Orchids produces the partial run ①, $\sigma_0 \xrightarrow{2}$ ② $[Pid := 57, Euid := 500, Tgt := 58]$, where $\sigma_0$ is the empty substitution.

Think of these partial runs as being *threads* running in parallel, of a single program that tries several ways of matching subflows against the signature (1). (Threads will actually be partial runs, plus some extra information, but we shall equate the two concepts for now.) Such threads will be put in a queue. Currently, this queue only contains thread ($i$) below (i.e., signature (1), at state ②), with substitution $[Pid := 57, Euid := 500, Tgt := 58]$, and the subflow of the corresponding partial run contains just 2. From now on, we write subflows with visible spaces ␣ to make explicit those events that were not taken into account; e.g., we write ␣2␣␣5 6 instead of $\{2, 5, 6\}$.

($i$) 

$[Pid := 57, Euid := 500, Tgt := 58]$ ␣2

In other words, ORCHIDS is considering event 2 as the first event of a possible attack.

Now ORCHIDS reads event 3, and decides to create a new thread (*ii*). Indeed, event 3 is also matched by the first pattern of the signature (1), so might also be the beginning of a possible attack. The current state of the thread queue is now:

(*i*)  $[Pid := 57, Euid := 500, Tgt := 58]$ ␣2␣

(*ii*)  $[Pid := 100, Euid := 500, Tgt := 101]$ ␣␣3

ORCHIDS has to *spawn* this new thread. Otherwise, it might miss an attack. If ORCHIDS had not spawned this new thread, there would be opportunities for intruders to launch so-called *masking attacks*. In other words, to start fake attack beginnings so as to lead the intrusion detection system on a false track. ORCHIDS cannot know whether there is indeed an attack starting at event 2 (first thread), or at event 3 (second thread), or none, but needs to consider both possibilities. Similar behavior is typical of modern multi-event intrusion detection systems, e.g., chronicles [4], or GnG [15].

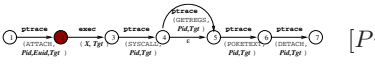
ORCHIDS now reads event 4, i.e., the `exec` event launching the instance of `modprobe` where the shellcode will eventually be inserted. The thread queue is now:

(*i*)  $[Pid := 57, Euid := 500, Tgt := 58]$ ␣2␣␣

(*ii*)  $[Pid := 100, Euid := 500, Tgt := 101, X := \text{"modprobe"}]$ ␣␣3 4

where the second thread has advanced to state ③, and is waiting on an event matching `ptrace`(`SYSCALL`, $Pid, Tgt$). The first thread does not advance, since the value of $Tgt$ (here, 101) does not match the one it already got (58).

If this seems natural to you, you have probably missed something—or you're clever. To avoid masking attacks, ORCHIDS should also have launched a third thread:

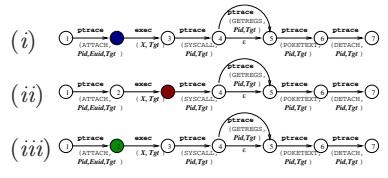
 $[Pid := 100, Euid := 500, Tgt := 101]$ ␣␣3 ␣

Indeed, it may be the case that the `exec` event 4 was only used to mount a masking attack again. If this is the case, we should spawn the thread above, which would disregard event 4 in the hope of finding a later `exec` event which would be the right one.

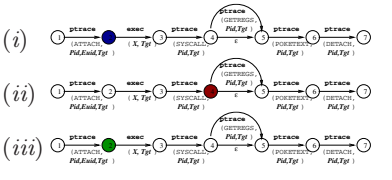
ORCHIDS does not spawn this thread, because it is able to show that this is useless. It is not that this thread has no chance of eventually detecting an attack: this would not be true. But, if this new thread eventually detects an attack, the corresponding subflow will never be shortest: if the new thread detects an attack at some event $n$, with subflow ␣␣3␣$\ldots n$, then thread (*ii*) will have detected an attack at event $n$ too, with subflow of the form ␣␣3 4 $\ldots n$. Now notice that the latter is strictly smaller in the $\preceq$ ordering, hence is more informative. We

can therefore safely ignore the above, useless, thread: we say that thread (*ii*) *subsumes* the above thread. This is an example of a *green cut*, see Section 5. Such green cuts are crucial to the efficiency of Orchids.

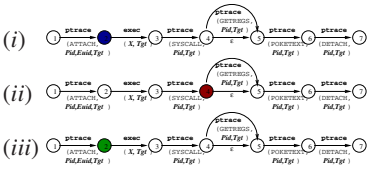
Orchids now reads event 5, which may against be the beginning of a `ptrace` attack. So Orchids launches a new thread:




Event 6 is irrelevant, and event 7 advances thread (`ii`):




Again, there is no need to create another thread that would consider the possibility that thread (`ii`) might not advance, because it would be subsumed by (`ii`), i.e., it would violate shortest runs. Event 8, `ptrace`(`SYSCALL`, 100, 101), matches the `ptrace`(`SYSCALL`, $Pid, Tgt$) pattern of the non-$\epsilon$-transition of thread (*ii*) from state ④ to ⑤. Again thanks to the shortest run trick, Orchids does not need to consider spawning a new copy of thread (*ii*) that would remain in state ④. More importantly, Orchids does not need to consider spawning a new copy of thread (*ii*) that would advance to state ⑤: again, Orchids detects that this would be subsumed. The thread queue is now therefore:




Reading event 9, Orchids decides to advance thread (*ii*) to state ⑥. Again, Orchids is able to show that it would be useless to spawn a copy of thread (*ii*) that would wait at state ⑤, because it would be subsumed. Orchids will then ignore events 10 and 11 in thread (*ii*), although they would be relevant (we let the reader rewrite the signature so that it captures *all* relevant `ptrace`(`POKETEXT`, ...) events), and will reach the final state ⑦ with thread (*ii*), and subflow ␣␣3 4␣␣7 8 9␣␣12.

At this point, Orchids' thread queue still contains threads (*i*) and (*iii*). These may be indicative of attacks starting at event 2, resp. 5, and which haven't been completed yet. For the moment, where we have just read event 12, Orchids reports an alert. We decided to have Orchids kill the offending user's processes

(with pid 100, and all descendants), and to close his account. Such retaliation measures have sometimes been described as characteristic of intrusion *prevention* systems, as opposed to intrusion detection systems. They are needed: a typical shellcode will insert some form of trapdoor into the system, such as setting the `setuid` bit on one of the user's process, to allow him to become root at any later time, without running the attack again.

The actual signature ORCHIDS uses to detect the `ptrace` attack is a bit more complicated. While ORCHIDS really reports and retaliates at state ⑦, this state is not final, and the real signature has added transitions. This allows ORCHIDS to track down all events done by the shellcode (with pid obtained in variable $Tgt$, here 101) until it exits (which it will eventually do, if only because ORCHIDS sent it a `KILL` signal). This allows a security engineer to analyze the inserted shellcode and its effects—this is called *forensic analysis*—and to take appropriate corrective countermeasures.

## 4 The Core Algorithm

The core algorithm that ORCHIDS uses, and which we have illustrated in Section 3.2, is based on the algorithm of [13, Figure 6]. However, the latter algorithm contains a bug, which the first author found 6 months after the paper was published. We take the opportunity to describe a correct algorithm, with a simpler presentation.

As far as simplifications go, first, we don't consider green cuts for now, in particular those related to shortest runs: see Section 5. Also, we consider only one signature $\Sigma = (Q, I, T, \Delta)$, although the extension to more is straightforward. Finally, we assume $\Sigma$ does not contain any $\epsilon$-transition. Removing $\epsilon$-transitions is done mostly as in standard finite-state automata, and only requires that we can form the conjunction $g_1 \wedge g_2$ of two guards $g_1$ and $g_2$, so that $[\![g_1 \wedge g_2]\!]\sigma$ is true if and only if $[\![g_1]\!]\sigma$ and $[\![g_2]\!]\sigma$ are both true: whenever $\Sigma$ contains two transitions $(q_1, p, g_1, q_2)$ and $(q_2, \epsilon, g_2, q_3)$, add the transition $(q_1, p, g_1 \wedge g_2, q_3)$ unless it is already present. When the signature is saturated under applications of this rule, remove all $\epsilon$-transitions.

The main idea of the algorithm is to keep the thread queue sorted, and to traverse this queue in such a way that the first thread with a given birthdate $i_1$ and signature $\Sigma$ that reaches a final state in the queue is shortest. Then, we remove all other threads with the same birthdate $i_1$ and $\Sigma$ from the queue—we *kill* these threads.

Intuitively, it should be enough to keep all threads sorted in the lexicographic ordering of the corresponding subflows, but this is wrong. Imagine the current thread queue contains threads corresponding to subflows 1 2 3, 1 2␣, and 1␣3. If event 4, the next event to be considered, led each of these to a final state, then we would have to pick the lexicographically smallest subflow among 1 2 3 4, 1 2␣4, and 1␣3 4: this is 1 2 3 4. Observe that $\{1,2,3,4\} <_{lex} \{1,2,4\} <_{lex} \{1,3,4\}$, where $<_{lex}$ is lexicographic ordering. Before event 4, we would therefore like the threads to be ordered as 1 2 3, then 1 2␣, then 1␣3. This way, no

reordering will have to happen when adding 4 to each subflow. However, 1 2 3 is certainly not smaller, lexicographically, than 1 2␣, i.e., $\{1, 2\}$! So we have to maintain the thread queue in some different ordering. This was recognized in [13, Theorem 4.11], where an ordering $<_i$ is defined for this purpose, for each event position $i$. (In the example, $i = 3$.) The right ordering is given by: for every subflows $D, D' \subseteq \{1, \ldots, i\}$, $D <_i D'$ if and only if $D$ and $D'$ have the same least element, and $D \cup \{i+1\} <_{lex} D' \cup \{i+1\}$. Roger and the first author [13] use a more complex, equivalent formula (up to the condition on least elements). Let $\leq_i$ be the reflexive closure of $<_i$, i.e., $D \leq_i D'$ if and only if $D = D'$ or $D <_i D'$.

Say that a list of partial runs $R_1, R_2, \ldots, R_m$ is $\leq_i$*-sorted* if and only if $R_j \leq_i R_k$ implies $j \leq k$. We aim at keeping queues of partial runs sorted, with minimal algorithmic effort. Whenever we read event number $i+1$, starting from a $\leq_i$-sorted thread queue $R_1, R_2, \ldots, R_k$, we must create a $\leq_{i+1}$-sorted thread queue of all possible *extensions* of the runs $R_j$, $1 \leq j \leq k$, as predicted by the semantics of signatures, and all possible partial runs *starting* at event $i+1$.

Extensions are defined as follows. Let $R = q_0, \sigma_0 \xrightarrow{i_1} q_1, \sigma_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} q_k, \sigma_k$ be a partial run, with subflow included in $\{1, \ldots, i\}$, and $R'$ a partial run with subflow included in $\{1, \ldots, i, i+1\}$. We say that $R'$ *extends* $R$ at position $i+1$ if and only if either $R' = R$ (wait without taking a transition), or $R' = q_0, \sigma_0 \xrightarrow{i_1} q_1, \sigma_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} q_k, \sigma_k \xrightarrow{i+1} q_{k+1}, \sigma_{k+1}$, where there is a transition $(q_k, p, g, q_{k+1}) \in \Delta$ with $p \neq \epsilon$, $\sigma_k \vdash p \triangleleft t_{i+1} \Rightarrow \sigma_{k+1}$, and with $[\![g]\!]\sigma_{k+1}$ true (go through the transition, acquiring new values for variables, and proceed to some later point in the event flow). In the latter case, $R'$ extends $R$ *non-trivially*, *through* the *outgoing* transition $(q_k, p, g, q_{k+1})$. Remember we assume $\Sigma$ does not contain any $\epsilon$-transition, so we can safely ignore them.

A partial run *starts* at event $i+1$ if and only if it is of the form $q_0, \sigma_0 \xrightarrow{i+1} q_1, \sigma_1$, where $q_0 \in I$, $\sigma_0$ is the empty substitution, and there is a transition $(q_0, p, g, q_1) \in \Delta$ with $\sigma_0 \vdash p \triangleleft t_{i+1} \Rightarrow \sigma_1$, and with $[\![g]\!]\sigma_1$ true.

Given a $\leq_i$-sorted list of partial runs $R_1, R_2, \ldots, R_m$, we must produce a $\leq_{i+1}$-sorted list $R'_1, R'_2, \ldots, R'_n$ of all partial runs extending some $R_j$, $1 \leq j \leq m$, or starting at $i+1$. In the case of extensions of the partial runs $R_j$, the idea of the algorithm of [13] is to enumerate each $R_j$ in turn, and to list the partial runs $R'_k$ that extend $R_j$, starting with those that extend $R_j$ non trivially. For example, starting from the partial runs 1 2 3 $<_3$ 1 2␣$<_3$ 1␣3 (where we identify partial runs with their subflows), imagine each has both trivial and non-trivial extensions at position 4. We start with 1 2 3, and output 1 2 3 4 first, then 1 2 3␣. Going on with 1 2␣, we output 1 2␣4, then 1 2␣␣. Eventually, this algorithm will output the partial runs 1 2 3 4, 1 2 3␣, 1 2␣4, 1 2␣␣, 1␣3 4, and 1␣3 ␣. We let the reader check that this is $\leq_4$-sorted.

However, there is a bug, which occurs whenever two partial runs are generated that induce the *same* subflow. Imagine for example that we must generate two partial runs with subflow 1␣3 4, on reading event 4. The above algorithm lists them in an arbitrary order. However, it may be that the first one will eventually lead to a complete run such as 1␣3 4␣6, and that the second one will lead to another complete run such as 1␣3 4 5 6… and 1␣3 4␣6, the first one, is then not shortest.

ORCHIDS uses a corrected algorithm, where partial runs are first grouped in *blobs*, i.e., non-empty sets of threads with the same subflow. Each blob therefore has a unique associated subflow. Then, blobs are $\leq_i$-sorted, in the sense that the associated subflows are $\leq_i$-sorted. In other words, a list of blobs $B_1, B_2, \ldots, B_m$ is $\leq_i$*-sorted* if and only if $D_j \leq_i D_k$ implies $j \leq k$, for all $1 \leq j, k \leq m$, writing $D_j$ for $B_j$'s subflow.

More precisely, at position $i$, ORCHIDS produces a $\leq_i$-sorted list $B_1, B_2, \ldots, B_m$. On reading event number $i+1$, ORCHIDS produces the queue described in Proposition 1 below, obtained by listing all partial runs starting at $i+1$ in a unique blob $B'_0$, and dealing with partial runs from $B_j$ by first listing all non-trivial extensions of partial runs from $B_j$, in a new blob $B'_{2j-1}$ that will precede the blob $B'_{2j}$ of the (unique) trivial extension. In other words, the corrected algorithm works as above, except it needs to consider blobs instead of single partial runs.

**Proposition 1.** *Let $B_1, B_2, \ldots, B_m$ be a $\leq_i$-sorted list of blobs, and assume all the subflows of each $B_j$, $1 \leq j \leq m$, are contained in $\{1, \ldots, i\}$. Let $B'_0$ be the set of all partial runs starting at $i+1$, $B'_{2j-1}$ be the set of all non-trivial extensions to partial runs in $B_j$, $B'_{2j}$ be the set of all trivial extensions to partial runs in $B_j$, $1 \leq j \leq m$. Then the queue obtained from $B'_0, B'_1, B'_2, \ldots, B'_{2m-1}, B'_{2m}$ by eliminating those $B'_j$s that are empty is $\leq_{i+1}$-sorted, and their subflows are contained in $\{1, \ldots, i, i+1\}$.*

*Proof.* Assume that $B'_0, B'_1, B'_2, \ldots, B'_{2m-1}, B'_{2m}$ is not $\leq_{i+1}$-sorted. Let $D'_j$ be the subflow of $B'_j$, for all $j$, and $D_j$ be the subflow of $B_j$. Then there are $j', k'$ with $0 \leq k' < j' \leq 2m$ and $D'_{j'} \leq_{i+1} D'_{k'}$. Note that $k' \neq 0$, since the birthdate of any partial run in $B'_0$ is $i+1$, which is different from all other birthdates. Write $k' = 2k - \delta_k$ and $j' = 2j - \delta_j$, where $\delta_k, \delta_j$ are 0 or 1, and $k \leq j$. If $k = j$, then $k' < j'$ implies $\delta_k = 1$, $\delta_j = 0$, so that $D'_{k'} = D_k \cup \{i+1\}$ (the partial runs of $B'_{k'} = B'_{2k-1}$ are non-trivial extensions of those of $B_k$), and $D'_{j'} = D_k$ (those of $B'_{j'} = B'_{2j} = B'_{2k}$ are trivial extensions). But $D_k \cup \{i+1\} <_{i+1} D_k$, so $D'_{k'} <_{i+1} D'_{j'}$, contradiction.

So $k < j$. Then $D_{k'}$ equals $D_k$, possibly with $i+1$ added, and $D_{j'}$ equals $D_j$, possibly with $i+1$ added. Since $B_1, B_2, \ldots, B_m$ is $\leq_i$-sorted, it is impossible that $D_j \leq_i D_k$, i.e., that $D_j \cup \{i+1\} \leq_{lex} D_k \cup \{i+1\}$. Since $\leq_{lex}$ is a total ordering, we must have $D_k \cup \{i+1\} <_{lex} D_j \cup \{i+1\}$. Write the elements of $D_k$ as $i_1 < i_2 < \ldots < i_p$ (with $i_p < i+1$), those of $D_j$ as $j_1 < j_2 < \ldots < j_q$ (with $j_q < i+1$, and $j_1 = i_1$). Let $i_{p+1} = i+1$, $j_{q+1} = i+1$. Since $D_k \cup \{i+1\} <_{lex} D_j \cup \{i+1\}$, for some $\ell$ between 1 and $\min(p+1, q+1)$, $i_1 = j_1$, $i_2 = j_2$, $\ldots$, $i_{\ell-1} = j_{\ell-1}$, and $i_\ell < j_\ell$. Now $\ell \neq p+1$, else $i+1 = i_\ell < j_\ell \leq j_{q+1} = i+1$. So $\ell \leq p$. But then $D_{k'} \cup \{i+2\}$, which is composed of $i_1, i_2, \ldots, i_p$ (optionally $i_{p+1} = i+1$) and $i+2$, is lexicographically smaller than $D_{j'} \cup \{i+2\}$, which is composed of $j_1, j_2, \ldots, j_q$ (optionally $j_{q+1} = i+1$) and $i+2$. That is, $D_{k'} <_{i+1} D_{j'}$, contradiction. □

While we have equated threads with partial runs until now, *threads* are in fact pairs of a partial run $R$ and an outgoing transition $(q_k, p, g, q_{k+1})$. One may think of a thread as *waiting* on a particular transition to fire. In general, there may

be several threads with the same partial run, waiting on different transitions in the same blob. From now on, call *thread queue* at position $i$ a $\leq_i$-sorted list of blobs, composed of such threads. At the moment, this organization of blobs in threads rather than in partial runs only leads to a minor modification in the core algorithm. This will become important in Section 5.

Additionally, Orchids maintains a set $Kill$ of birthdates of partial runs that have reached their final state, to kill non-shortest runs. On reading event $i+1$, Orchids first resets $Kill$ to $\emptyset$. Orchids runs through the threads $R$ in $B_1, B_2, \ldots, B_m$ as described in Proposition 1, with two modifications. First, whenever a thread with run $R'$ is produced in one of the new blobs $B'_{j'}$, $0 \leq j' \leq 2m$, that reaches a final state, Orchids adds the birthdate $i_1$ of $R'$ to $Kill$. This is a shortest complete run. Second, Orchids kills all other threads with the same birthdate $i_1$ by simply ignoring the threads in $B_1, B_2, \ldots, B_m$ whose birthdate are in $Kill$ when their turn comes.

Orchids also ignores a number of other threads, see Section 5. Note that the actual thread queue, consisting of subsets of the blobs of Proposition 1, will also remain $\leq_i$-sorted at each event number $i$, guaranteeing that the unique complete run that will reach a final state (with given birthdate and signature) indeed has a shortest subflow.

Finally, we didn't say what Orchids did on reaching a final state. It might seem obvious that this would be the right point to emit a report, warning the security administrator that an attack has just successfully completed, and to take active countermeasures. This is in fact wrong, and confuses two roles for final states. One of these roles is recognizing that enough information has been collected to conclude that some attack was indeed under way. The other role is to terminate Orchids monitoring, and kill the corresponding threads. These two roles are distinct. The actual signature we use for `ptrace` has more states. State ⑦ is not final, and is the state at which Orchids takes corrective actions—here, Orchids will emit an attack report, store it into a secured database of successful fatal attacks, kill the offending attacking process (whose pid is in $Pid$) and all its descendants, securely close the attacker's account (whose id is in $Euid$) through an SSH connection to the attacked machine. (We assume that Orchids runs on a different, dedicated host, for obvious security reasons.) However, killing subprocesses and closing user accounts takes some time, in particular if this is done through a remote SSH connection, so the shellcode has some time to do harm. The actual `ptrace` signature we use in Orchids has additional states following ⑦, whose purpose is to trace and record all subsequent events done by the shellcode. This allows later, precise forensic analysis of the attack, and is crucial both for repairing the attacked host and for acquiring information on emerging viruses and worms.

## 5 Cuts, Green Cuts, Red Cuts

By *cut*, we mean any optimization or construction allowing one to kill threads. Cuts are important to be able to bound the number of active threads at any

given position in the event flow. Following Prolog conventions [1], we distinguish between *green cuts*, which preserve the semantics, and *red cuts*, which don't. We first describe green cuts based on the notion of monotonic variables. These cuts are green, because they eliminate threads that will provably *never* reach a final state. Some other green cuts allow one to kill threads that may reach a final state, but if they do, the corresponding subflow will never be shortest. We have already seen an example of this in Section 3.2. We explain this in a second subsection. We argue for red cuts in the final subsection.

**Green Cuts I: Monotonicity and Generalized Timeouts.** The *monotonicity* cuts we describe now are typically justified by the need for *timeouts*, although they are not limited to the latter. Timeouts are needed to eliminate proliferating threads. Otherwise, attacker might mount denial-of-service attacks against the intrusion prevention system itself. Instead, it is necessary to kill threads that have exceeded a certain quota in terms of time or number of events. Naturally, this opens the door to *slow attacks*, i.e., to attacks that would evade detection by taking a long time to complete, and by generating events that are far away from each other in the event flow. A security administrator has to define suitable timeouts, as a result of a compromise between avoiding denial-of-service attacks and detecting slow attacks.

Enough freedom should be given to the security administrator to tune timeout information. We said that timeout information may be some combination of elapsed time and number of events. We may also take into account other time fields: the time at which a given event happened on a remote host, the time at which it was sent to the intrusion prevention system, the time at which it was received, the time at which it was logged. These are usually available as different time fields in the incoming events.

Instead of designing a specific notation for timeouts, it is simpler and more versatile to just use the guards $g \in \mathcal{G}$ for this purpose. For example, assuming the rigid variable $T_0$ holds the time at which the first event in the current partial run was logged and $I_0$ holds its position, the flexible variable $\$t$ holds the time at which the current event was logged, and the flexible variable $\$i$ holds the event position (obtained through pattern-matching), the guard $\$t < T_0 + 60 \wedge \$i < I_0 + 30\,000$ states that we wish to continue to monitor the given possible attack for at most 60 seconds and at most 30 000 events.

Such guards by themselves are not enough to reduce the number of threads. However, recognizing that a guard will always be false in the future allows us to disregard it entirely. We accomplish this in ORCHIDS by subdividing the `int` type of integers (and other numerical types) into those of values that are *monotonic* (non-decreasing over time), *antitonic* (non-increasing over time), *constant* (i.e., both monotonic and antitonic), and *arbitrary*. We also equate Boolean values as the subtype consisting of 0 (false) and 1 (true) for this purpose. Such monotonicity information can be formalized by using the familiar 4-element lattice **Four** of subsets of $\{\uparrow, \downarrow\}$ ordered by inclusion: $\emptyset$ means arbitrary, $\{\uparrow\}$ monotonic, $\{\downarrow\}$ antitonic, and $\{\uparrow, \downarrow\}$ means constant. Numerical types $\tau_i$ also include a monotonicity information, in **Four**, e.g., $\texttt{int}/\{\downarrow\}$.

Specific fields in events are marked as monotonic, such as time fields, or event numbers. Formally, each function symbol $f$ comes with a typing rule, e.g., stating that any term $f(t_1, \ldots, t_n)$ gives each $t_i$ some type $\tau_i$, $1 \leq i \leq n$. To simplify the presentation, assume that all variables have type $\texttt{int}/m$ for some monotonicity information $m$, and that the types $\tau_i$ mentioned earlier are either $\texttt{int}/\emptyset$, or $\texttt{int}/\{\uparrow\}$—in which case we say that $i$ is a *monotonic position*. Given a pattern $p$ and a set $V$ of variables (denoting variables that are already bound to some value), let $\Gamma[p, V]$ be the typing context of all bindings $x : \tau$, where either $x$ is rigid and in $V$ and $\tau = \texttt{int}/\{\uparrow, \downarrow\}$ (rigid variables, once bound, will remain constant), or $x$ is flexible and occurs in $p$ at some monotonic position, and $\tau = \texttt{int}/\{\uparrow\}$. Guards are typed using typing rules that include:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{m \supseteq m'}{\texttt{int}/m <: \texttt{int}/m'} \qquad \frac{}{\tau <: \tau} \qquad \frac{\Gamma \vdash t : \tau \quad \tau <: \tau'}{\Gamma \vdash t : \tau'}$$

$$\frac{(c \text{ numerical constant})}{\Gamma \vdash c : \texttt{int}/\{\uparrow, \downarrow\}} \qquad \frac{\Gamma \vdash t_1 : \texttt{int}/m_1 \quad \Gamma \vdash t_2 : \texttt{int}/m_2}{\Gamma \vdash t_1 + t_2 : \texttt{int}/(m_1 \cap m_2)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{int}/m_1 \quad \Gamma \vdash t_2 : \texttt{int}/m_2}{\Gamma \vdash t_1 < t_2 : \texttt{int}/(\overline{m_1} \cap m_2)} \qquad \frac{\Gamma \vdash t_1 : \texttt{int}/m_1 \quad \Gamma \vdash t_2 : \texttt{int}/m_2}{\Gamma \vdash t_1 \wedge t_2 : \texttt{int}/(m_1 \cap m_2)}$$

In the last rules, we use the fact that Boolean values are considered as integers, and we take the convention that $\overline{\uparrow} = \downarrow$, $\overline{\uparrow} = \downarrow$, $\overline{m} = \{\overline{s} \mid s \in m\}$. Using the typing rules above, it is easy to see that we can derive $\$t : \texttt{int}/\{\uparrow\}, \$i : \texttt{int}/\{\uparrow\}, T_0 : \texttt{int}/\{\uparrow, \downarrow\}, I_0 : \texttt{int}/\{\uparrow, \downarrow\} \vdash \$t < T_0 + 60 \wedge \$i < I_0 + 30\ 000 : \texttt{int}/\{\downarrow\}$. This implies that the guard $g = (\$t < T_0 + 60 \wedge \$i < I_0 + 30\ 000)$ is antitonic, in particular that if it is false at event position $i$, it will remain false at every later position. This is a consequence of the following, easily proved proposition, with $t = g$. We assume the evaluation function $[\![\_]\!]\sigma$ to behave as expected, e.g., $[\![t_1 + t_2]\!]\sigma = [\![t_1]\!]\sigma + [\![t_2]\!]\sigma$.

**Proposition 2.** *Assume that event fields are marked monotonic or arbitrary, and that monotonic fields of events $t_i$ are integers that are non-decreasing in $i$. For any pattern $p$, substitution $\sigma$, and term $t$, if $\Gamma[p, \mathrm{dom}\ \sigma] \vdash t : \texttt{int}/m$ is derivable, and if $\sigma \vdash p \triangleleft t_i \Rightarrow \sigma_i$, then for every $j > i$ such that $\sigma \vdash p \triangleleft t_j \Rightarrow \sigma_j$, $[\![t_i]\!]\sigma_i$ and $[\![t_j]\!]\sigma_j$ are integers, $[\![t_i]\!]\sigma_i \leq [\![t_j]\!]\sigma_j$ if $\uparrow \in m$, and $[\![t_i]\!]\sigma_i \leq [\![t_j]\!]\sigma_j$ if $\downarrow \in m$.*

Orchids implements this as follows. Recall that one may think of each thread, with partial run $R = q_0, \sigma_0 \xrightarrow{i_1} q_1, \sigma_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} q_k, \sigma_k$, as *waiting* on a transition $(q_k, p, g, q_{k+1})$ to fire. If $\sigma_k \vdash p \triangleleft t_i \Rightarrow \sigma'$ for some substitution $\sigma'$ but $[\![g]\!]\sigma' = 0$ (false), and if $\Gamma[p, \mathrm{dom}\ \sigma_k] \vdash g : \texttt{int}/\{\downarrow\}$ is derivable, then Orchids kills the thread, i.e., removes it from its blob (and removes the blob from the queue if it becomes empty): not only does this transition fail to fire at position $i$, it will *never* fire.

**Green Cuts II: Predicting Non-Shortest Runs.** The algorithm of Section 4 kills all threads with a given birthdate and signature, once a corresponding (shortest) complete run has been found. However, as we have illustrated in Section 3.2, Orchids also kills some threads *in advance*, knowing that they cannot be completed to a shortest run. This is crucial to the performance of

ORCHIDS. Otherwise, we would accumulate useless threads, only to kill them en masse when one of them completes, if ever.

Returning to the example of Section 3.2, the first case this happened was on reading event 4, where we decided that it was useless to spawn the thread with subflow $\_\_3\_$, since it would be subsumed by thread (*ii*), with subflow $\_\_3$ 4. In this example, most states only have one outgoing transition, so that we had only one thread per partial run. Hence, we equated threads with partial runs. In general, we should be careful that threads are partial runs *waiting* on a given transition. Here, before reading event 4, we had a thread (*ii*), with a partial run of subflow $\_\_3$ $\_$, waiting on transition $(②, \texttt{exec}(X, Tgt), 1, ③)$. The core algorithm should in principle create two threads from the latter when reading event 4 ($\texttt{exec}(\texttt{"modprobe"}, 101)$). One would advance this thread to one with subflow $\_\_3$ 4, waiting on $(③, \texttt{ptrace}(\texttt{SYSCALL}, Pid, Tgt), 1, ④)$. The other (the trivial extension of the partial run) would decide to continue waiting on a later event matching $\texttt{exec}(X, Tgt)$. As we have already argued, the latter is useless.

It would be wrong to think that all trivial extensions of a partial run $R$ are subsumed by any non-trivial extension of $R$, i.e., that it is always useless to wait when a transition could be fired. Although the case does not happen in the example of Section 3.2, consider the signature ① —start (*Tgt*)→ ② —action (*Tgt, X*)→ ③ —final (*X*)→ ④, and the event flow 1: `start` (58); 2: `action` (58, `A`); 3: `action` (58, `B`); 4: `final` (`B`). On reading event 2, while in state ②, both the trivial extension $1\_$ (with $Tgt := 58$) and the non-trivial extension 1 2 (with $Tgt := 58$, $X := \texttt{A}$) have to be considered. The point is that we don't know whether `A` is the right value for $X$ that will lead to a subflow that matches the signature. And indeed, the only such subflow is $1\_3$ 4, with $Tgt := 58, X := \texttt{B}$. We may say that, at state ②, the value of $X$ still has to be discovered. On the contrary, in the `ptrace` example, the value of rigid variable $Tgt$ has already been discovered at state ②, while the value of $X$, which we discover at this point, will never be used by any later guard.

This is formalized as follows. Given a computable property $P(\Gamma, g)$ of a typing context $\Gamma$ and a guard $g$, we say that $P$ *holds at all reachable transitions* from $\Gamma$ and the transition $(q, p, g, q')$, inductively, if and only if $P(\Gamma, g)$ holds and $P$ holds at all reachable transitions from $\Gamma \oplus \{x : \texttt{int}/\{\uparrow, \downarrow\} \mid x \in \mathrm{fv}(p')\}$ and $(q', p', g', q'')$ for all outgoing transitions $(q', p', g', q'')$. This is meant to say that $P(\Gamma', g')$ holds whenever we reach a transition $(q', p', g', q'')$, where $\Gamma'$ is $\Gamma$, with all variables bound in-between assumed constant. Let $\Gamma\langle p, V\rangle$ be the typing context of all bindings $x : \tau$, where: if $x$ is in $V$ and not a flexible variable in $\mathrm{fv}(p)$, then $\tau = \texttt{int}/\{\uparrow, \downarrow\}$; if $x$ is flexible and occurs in $p$ at some monotonic position, then $\tau = \texttt{int}/\{\uparrow\}$; otherwise, $\tau = \texttt{int}/\emptyset$.

**Proposition 3.** *Assume that event fields are marked monotonic or arbitrary, and that monotonic fields of events $t_i$ are integers that are non-decreasing in $i$.*

*Let $R = q_0, \sigma_0 \xrightarrow{i_1} q_1, \sigma_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} q_k, \sigma_k$ be a partial run, with subflow included in $\{1, \ldots, i\}$, and where $q_k$ is not final. Let $(q_k, p, g, q_{k+1})$ be some outgoing transition, assume that $\sigma_k \vdash p \triangleleft t_{i+1} \Rightarrow \sigma_{k+1}$, and that $P$ holds at all reachable transitions from $\Gamma\langle p, \mathrm{dom}\ \sigma_k\rangle$ and $(q_k, p, g, q_{k+1})$, where $P(\Gamma', g')$ is the property: $\Gamma' \vdash g' : \texttt{int}/\{\downarrow\}$*

*is derivable. For any complete run* $q_0, \sigma_0 \xrightarrow{i_1} q_1, \sigma_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} q_k, \sigma_k \xrightarrow{i_{k+1}} q_{k+1}, \sigma'_{k+1} \xrightarrow{i_{k+2}} \ldots \xrightarrow{i_m} q_m, \sigma'_m$ *with* $i+1 < i_{k+1}$*, there is a strictly shorter complete run* $q_0, \sigma_0 \xrightarrow{i_1} q_1, \sigma_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} q_k, \sigma_k \xrightarrow{i+1} q_{k+1}, \sigma_{k+1} \xrightarrow{i_{k+2}} \ldots \xrightarrow{i_m} q_m, \sigma_m$.

*Proof.* (Sketch.) I.e., we can build a strictly shorter run by firing the transition $(q_k, p, g, q_{k+1})$ at event position $i+1$ instead of waiting for some later position $i_{k+1}$. The assumption $\sigma_k \vdash p \triangleleft t_{i+1} \Rightarrow \sigma_{k+1}$ ensures that we can indeed fire this at position $i+1$. All further transitions, from $q_{k+1}$ to $q_{k+2}$ to … to $q_m$ are the same in both runs. This may change variable bindings, from $\sigma'_{k+1}$ to $\sigma_{k+1}$, …, and from $\sigma'_m$ to $\sigma_m$. But, by the typing condition, this can only make the value of guards increase. Since all guards were made true by $\sigma'_{k+1}$, …, $\sigma'_m$, the same guards are made true by $\sigma_{k+1}$, …, $\sigma_m$. □

So, under the assumptions of Proposition 3, it is safe to advance along the transition $(q_k, p, g, q_{k+1})$, *without* spawning a thread waiting on a later event position for the same transition. Note that, as a particular case, the assumptions of Proposition 3 are satisfied whenever $p$ only binds rigid variables, and those that were not in dom $\sigma_k$ are free in no guard occurring later in the signature. This is what we illustrated in Section 3.2.

Such green cuts are, as we have said, crucial to the performance of Orchids. Totel *et al.* have already recognized the importance of timeouts, and shown [15, Figure 7] that with a timeout value of 1 s on a user machine, the number of plans (similar to our threads) culminated to a few hundreds. We worked together in 2002, in the framework of the French RNTL project DICO, and evaluated our respective algorithms by comparing numbers of threads throughout several event flows, both artificial and real. On E. Totel's main signature example, which was meant to test the limits of multi-event intrusion detection systems, our algorithm never maintained more than 6 threads in the queue. Our worst case was on a real flow of 31 467 events, in which we had introduced two attack subflows, one on `sendmail` and one on `rpcinfo`, with interleaved events, and very far apart: our algorithm culminated to 19 threads, with an average of 7.1. (We didn't rely on any timeout.) Analysis showed that this good performance was a direct consequence of the green cuts described here, which allow us to kill threads that will violate shortest runs, *in advance*. We have also run Orchids on the LSV network, in normal operation, during six months in 2005. It caught some attack attempts (mostly IP range probing), while only consuming a few minutes of system time total.

**Red Cuts.** By red cut, we mean any feature designed to kill threads arbitrarily, possibly missing some attacks. This is in analogy with Prolog's cut "!" [1]. While the concept may seem ugly, it is definitely required in practice. For example, remember that the complete `ptrace` signature has more transitions after state ⑦ than shown in (1), which collect actions done by the intruder before its processes are killed and its account closed. On reaching state ⑦, we use a red cut to instruct Orchids to kill all threads with the same birthdate and signature.

So we collect intruder actions for only one instance of the attacks that succeeded with the same birthdate.

Red cuts are important, in general, to avert denial-of-service attacks against the intrusion prevention system. They allow a more direct control on the number of generated threads. Notably, they allow us to implement a form of the **Without** operator of Totel *et al.* [15], an effective tool to control the growth of the thread queue. This allows one to monitor one signature $\Sigma_1$, provided some other signature $\Sigma_2$ does not match in-between. For example, this allows us to monitor what a given process does, while it is alive, i.e., while no `exit` event is recorded by this process. This is implemented in ORCHIDS by running threads that monitor both $\Sigma_1$ and $\Sigma_2$. Once a complete run for $\Sigma_2$ is found, a red cut is issued that kills all pending instances of both $\Sigma_2$ and $\Sigma_1$.

## 6 Detecting Families of Attacks

The `ptrace` attack example may give the wrong impression that ORCHIDS requires one signature for each attack, requiring high maintenance overhead. Instead, one may write signatures that detect attacks by their effects, e.g., illicitly acquiring root privileges. In ORCHIDS, this is done by using the `pidtracker` signature, which has three states. There are transitions labeled with patterns matching calls to `fork`, `vfork`, `execve`, `setgid32`, and `setresuid32`, from state ① to ①. The latter two primitives are the only ones (in Linux) that may legitimately change one of the variants of the user id (i.e., user id, effective user id, saved user id), and these transitions are used to track all changes to these user ids by a given process (whose pid is stored in some rigid variable *Pid*). The first three primitives are monitored to track down all created processes. Even without red cuts, the shortest run semantics guarantees that no such call will be missed.

Additionally, there is a transition from ① to ②, tracking all calls made by the process with pid *Pid* with some user id different from the one obtained through tracking the events above. This detects any system call done with an unexpected user id, typically with user id 0 (root) while the process was normally running under a non-root user id. Finally, there is a transition from ① to ③, tracking calls to `exit` by process *Pid*, with a red cut to kill all threads monitoring the same process *Pid*.

The `pidtracker` rule is a practical way to detect instances of the `do_brk` attack [14], a vicious local-to-root attack in which the intruder repeatedly calls `do_brk` to allocate all available memory until kernel space is mapped into user space, and the process rights table in kernel space is modified to obtain root privileges. This is vicious in the sense that the only characteristic events of this attack are calls to `do_brk`, and a flurry of `SIGSEGV` signals. None of these are logged in any event logging system, for technical reasons. So a characteristic event subflow for this attack would be empty! And the `do_brk` attack has already been used, with disastrous effects: crackers used it to infect the Savannah servers (the master servers of the GNU distribution) and the Debian Linux master servers

in 2004. Once the infection was suspected, these servers had to be turned down for manual inspection of all packages, and this took several weeks.

The `pidtracker` signature detects that an attack such as `do_brk` has succeeded as soon as the offending process invokes the next system call. An added benefit of this signature is that it will detect *all* such local-to-root attacks. It is important that signatures be able to detect whole *families* of attacks, to decrease the maintenance overhead of the signature base. In this case, we discovered in 2005 that the `pidtracker` signature, unchanged, would catch the newer but similar `map`, `munmap`, `mremap` attacks.

This also shows that, contrary to popular belief, it is possible to detect *zero-day attacks*, i.e., attacks which are launched before an advisory is made public. To wit, the `pidtracker` rule caught the recent Linux `vmsplice` attack [12], an attack published less than two months before our presentation at RV'08. (By the way, this attack gets completely undetected by the SELinux security enhancement to Linux, under standard reference policies, in `ENFORCED` mode.)

## 7 Conclusion

There is much more that could be said about ORCHIDS, in particular from a perspective more geared towards security administrators. One strand would have been to expand on the fact that ORCHIDS is both able to detect bad behavior (attacks described through signatures), or deviation from good behavior (where "good" is defined through some security policy, of which the setuid model of Section 6 was a simple example). Another would have been to show how ORCHIDS, originally a misuse detection system, can also work as an anomaly detection system: adding statistical classifiers to ORCHIDS is essentially a matter of adding an event logging module that outputs statistical data in the form of events, which ORCHIDS can then match. We could also have demonstrated how ORCHIDS detects complex, *network* attacks such as the `mod_ssl` attack [3], using such a statistical module [8]. Instead, we have chosen to center our presentation on algorithmic issues, taking the opportunity to describe both the core algorithm and the notion of cuts—in particular the *green* cuts that are so central to the efficiency of ORCHIDS—in as clear and intuitive a way as possible. We hope to have demonstrated how efficient multi-event intrusion prevention was possible, and how much monitor technology was relevant to this task. ORCHIDS is freely available under the Cecill 2 (GPL) license [6].

## References

1. Clocksin, W., Mellish, C.: Programming in Prolog. Springer, Heidelberg (1981)
2. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1991)
3. McDonald, J., A.L. Digital Ltd., The Bunker: OpenSSL SSLv2 malformed client key remote buffer overflow vulnerability (July 2002), `http://www.securityfocus.com/bid/5363`

4. Morin, B., Debar, H.: Correlation of intrusion symptoms: An application of chronicles. In: Vigna, G., Krügel, C., Jonsson, E. (eds.) RAID 2003. LNCS, vol. 2820, pp. 94–112. Springer, Heidelberg (2003)
5. Morin, B., Mé, L., Debar, H., Ducassé, M.: M2D2: A formal data model for IDS alert correlation. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516. Springer, Heidelberg (2002)
6. Olivain, J.: ORCHIDS—real-time event analysis and temporal correlation for intrusion detection in information systems (2004), `http://www.lsv.ens-cachan.fr/orchids/`
7. Olivain, J., Goubault-Larrecq, J.: The Orchids intrusion detection tool. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 286–290. Springer, Heidelberg (2005)
8. Olivain, J., Goubault-Larrecq, J.: Detecting subverted cryptographic protocols by entropy checking. Research Report LSV-06-13, Laboratoire Spécification et Vérification, ENS Cachan, France, 19. pages (June 2006)
9. Pouzol, J.-P., Ducassé, M.: Formal specification of intrusion signatures and detection rules. In: Cervesato, I. (ed.) 15th IEEE Computer Security Foundations Workshop (CSFW 2002), pp. 64–76. IEEE Comp.Soc.Press, Los Alamitos (2002)
10. Purczyński, W.: Linux ptrace/execve race condition vulnerability. BugTraq Id 2529 (March 2001), `http://www.securityfocus.com/bid/2529`
11. Purczyński, W.: Linux kernel privileged process hijacking vulnerability. BugTraq Id 7112 (March 2003), `http://www.securityfocus.com/bid/7112`
12. Purczyński, W., qaaz.: Linux kernel prior to 2.6.24.2 'vmsplice_to_pipe()' local privilege escalation vulnerability (February 2008), `http://www.securityfocus.com/bid/27801`
13. Roger, M., Goubault-Larrecq, J.: Log auditing through model checking. In: 14th IEEE Computer Security Foundations Workshop (CSFW 2001), pp. 220–236. IEEE Computer Society Press, Los Alamitos (2001)
14. Starzetz, P.: Linux kernel 2.4.22 do_brk() privilege escalation vulnerability, K-Otik ID 0446, CVE CAN-2003-0961 (December 2003), `http://www.k-otik.net/bugtraq/12.02.kernel.2422.php`
15. Totel, E., Vivinis, B., Mé, L.: A language driven IDS for event and alert correlation. In: Deswarte, Y., Cuppens, F., Jajodia, S., Wang, L. (eds.) Security and Protection in Information Processing Systems, IFIP 18th World Computer Congress, TC11 19th International Information Security Conference, pp. 209–224. Kluwer, Dordrecht (2004)