On the complexity of monitoring Orchids signatures

Jean Goubault-Larrecq
Jean-Philippe Lachance

LSV, ENS Paris-Saclay, France
Coveo Solutions, Québec City, Québec, CA

September 28, 2016
An intrusion detection system, initially based on model-checking ideas [GL, Roger, CSFW‘01] semantics and optimizations made precise in [GL, Olivain, RV’08]

Should really be seen as a trace-based monitor.

Detection can be expensive (at least in theory).

Detection cost depends on signatures (=rules, =specifications):
Which signatures are expensive?
Can we decide which, algorithmically?
Complexity of monitors

* **Goal:** Estimate upper bound $f_S(n)$ on number of threads (=monitor instances) — after reading $n$ events — while monitoring signature $S$

  We shall give a simple **definition** of $f_S(n)$.

* **Decision problem:**
  INPUT: a signature $S$
  QUESTION: is $f_S(n)$ asymptotically polynomial?

  [Bonus question: if so, for which $k$ is it $O(n^k)$?]

  We give a **linear-time algorithm**.
Related work

- Complexity of monitor algorithms:
  - **RV-Monitor** [Luo et al., RV’14], data-driven:
    \( f_s(n) \) polynomial, degree \( k = \# \text{parameters} \)
  - **MonPoly-*** [Basin et al., JACM’15]:
    \( f_s(n) \) polynomial, degree \( k \): see paper

- We will reduce our problem to asymptotics of recurrence equations:
  see **Analytic Combinatorics** [Flajolet, Sedgwick ’09], but:
  — Our solution is more elementary
  — Our problem is slightly outside the scope of AC (max operator)
Intrusion detection through model-checking [JGL&Roger, CSF01]

- The monitored machines collect **events**:

```
open("/etc/passwd", "r", pid=58, euid=500)
ptrace (ATTACH, pid=57, euid=500, 58)
ptrace (ATTACH, pid=100, euid=500, 101)
exec (prog="modprobe", pid=101)
ptrace (ATTACH, pid=100, euid=500, 101)
exit (pid=58)
ptrace (SYSCALL, pid=100, 101)
ptrace (GETREGS, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (POKETEXT, pid=100, 101)
ptrace (DETACH, pid=100, 101)
```

- **Signatures** describe attacks:

```
1 2 3 4 5 6 7
ptrace (ATTACH, Pid,Euid,Tgt )
exec ( Tgt )
ptrace (SYSCALL, Pid,Tgt )
ptrace (GETREGS, Pid,Tgt )
ptrace (POKETEXT, Pid,Tgt )
ptrace (DETACH, Pid,Tgt )
```
rule pidtrack synchronized($pid)
{
  state init
  {
    expect (.auditd.syscall==SYS_clone)
      goto newpid;
  }

  state newpid! { 
    $pid = .auditd.exit;
    $uid = .auditd.euid;
    $gid = .auditd.egid;
    goto wait;
  }

  state wait!
  {
    expect (.auditd.pid == $pid &&
      .auditd.syscall == SYS_execve &&
      (.auditd.uid != .auditd.euid || .auditd.gid != .auditd.egid) &&
      .auditd.success == "yes")
      goto update_uid_gid;
    expect (.auditd.pid == $pid &&
      .auditd.syscall == SYS_setresuid ||
      .auditd.syscall == SYS_setresuid ||
      .auditd.syscall == SYS_setuid) &&
      .auditd.success == "yes")
      goto update_setuid;
    expect (.auditd.pid == $pid &&
      .auditd.syscall == SYS_setregid ||
      .auditd.syscall == SYS_setresgid ||
      .auditd.syscall == SYS_setgid) &&
      .auditd.success == "yes")
      goto update_setgid;
    expect (.auditd.pid == $pid &&
      .auditd.syscall == SYS_exit)
      goto end;
    expect (.auditd.pid == $pid &&
      (.auditd.euid != $uid || .auditd.egid != $gid))
      goto alert;
  }
}

state update_uid_gid!
{
  $uid = .auditd.euid;
  $gid = .auditd.egid;
  goto wait;
}

state update_setuid!
{
  case (.auditd.egid != $gid) goto alert;
  else goto update_uid_gid;
}

state update_setgid!
{
  case (.auditd.euid != $uid) goto alert;
  else goto update_uid_gid;
}

state alert!
{
  $newuid = .auditd.euid;
  $newgid = .auditd.egid;
  report();
}

state end!
{
  /* all went well */
}
Orchids signatures: checking

Create new thread (=monitor instance), waiting for some event satisfying Boolean condition

```
Orchids signatures: checking

rule pidtrack synchronized($pid)
{
    state init
    expect (.auditd.syscall==SYS_clone)
    goto newpid;
}

state newpid!

$pid = .auditd.exit;
$uid = .auditd.euid;
$gid = .auditd.egid;
goto wait;

state wait!
{
    expect (.auditd.pid == $pid &&
            .auditd.syscall == SYS_execve &&
            (.auditd.uid != .auditd.euid || .auditd.gid != .auditd.egid) &&
            .auditd.success == "yes")
    goto update_uid_gid;
    expect (.auditd.pid == $pid &&
            .auditd.syscall == SYS_setresuid ||
            .auditd.syscall == SYS_setreuid ||
            .auditd.syscall == SYS_setuid) &&
            .auditd.success == "yes")
    goto update_setuid;
    expect (.auditd.pid == $pid &&
            .auditd.syscall == SYS_setresgid ||
            .auditd.syscall == SYS_setregid ||
            .auditd.syscall == SYS_setgid) &&
            .auditd.success == "yes")
    goto update_setgid;
    expect (.auditd.pid == $pid &&
            .auditd.syscall == SYS_exit)
    goto end;
    expect (.auditd.pid == $pid &&
            (.auditd.euid != $uid || .auditd.egid != $gid))
    goto alert;
}

state update_uid_gid!
{
    $uid = .auditd.euid;
    $gid = .auditd.egid;
    goto wait;
}

state update_setuid!
{
    case (.auditd.egid != $gid) goto alert;
    else goto update_uid_gid;
}

state update_setgid!
{
    case (.auditd.euid != $uid) goto alert;
    else goto update_uid_gid;
}

state alert!
{
    $newuid = .auditd.euid;
    $newgid = .auditd.egid;
    report();
}

state end!
{
    /* all went well */
}

If found, fork new thread going to next state
Orchids signatures: checking

goto next state, waiting on 5 Boolean conditions: create 5 threads, one for each
Orchids signatures: checking

rule pidtrack synchronized($pid)
{
    state init
        expect (.auditd.syscall == SYS_clone)
    goto newpid;

    state newpid!
        $pid = .auditd.exit;
        $uid = .auditd.euid;
        $gid = .auditd.egid;
    goto wait;

    state wait!
        expect (.auditd.pid == $pid &&
                .auditd.syscall == SYS_execve &&
                (.auditd.uid != .auditd.euid || .auditd.gid != .auditd.egid) &&
                .auditd.success == "yes")
    goto update_uid_gid;
        expect (.auditd.pid == $pid &&
                .auditd.syscall == SYS_setresuid &&
                .auditd.success == "yes")
    goto update_setuid;
        expect (.auditd.pid == $pid &&
                .auditd.syscall == SYS_setresgid &&
                .auditd.success == "yes")
    goto update_setgid;
        expect (.auditd.pid == $pid &&
                .auditd.syscall == SYS_setgid &&
                .auditd.success == "yes")
    goto wait;

    state update_uid_gid!
        $uid = .auditd.euid;
        $gid = .auditd.egid;
    goto wait;

    state update_setuid!
        case (.auditd.egid != $gid) goto alert;
        else goto update_uid_gid;

    state update_setgid!
        case (.auditd.euid != $uid) goto alert;
        else goto update_uid_gid;

    state alert!
        $newuid = .auditd.euid;
        $newgid = .auditd.egid;
    report();

    state end!
        /* all went well */
    goto end;
}

lundi 26 septembre 16
Orchids signatures, as automata

rule pidtrack synchronized($pid)
{
  state init
  expect (.auditd.syscall==SYS_clone)
    goto newpid;
  state newpid!
    $pid = .auditd.exit;
    $uid = .auditd.euid;
    $gid = .auditd.egid;
    goto wait;
  state wait!
  expect (.auditd.pid == $pid &&
    .auditd.syscall == SYS_execve ||
    (.auditd.uid != .auditd.euid || .auditd.gid != .auditd.egid) &&
    .auditd.success == "yes")
    goto update_uid_gid;
  expect (.auditd.pid == $pid &&
    (.auditd.syscall == SYS_setresuid ||
    .auditd.syscall == SYS_setregid) &&
    .auditd.success == "yes")
    goto update_setuid;
  expect (.auditd.pid == $pid &&
    (.auditd.syscall == SYS_setresgid ||
    .auditd.syscall == SYS_setgid) &&
    .auditd.success == "yes")
    goto update_setgid;
  expect (.auditd.pid == $pid)
    goto end;
  expect (.auditd.pid == $pid &&
    (.auditd.euid != $uid || .auditd.egid != $gid))
    goto alert;

  state update_uid_gid!
    $uid = .auditd.euid;
    $gid = .auditd.egid;
    goto wait;

  state update_setuid!
    case (.auditd.egid != $gid) goto alert;
    case (.auditd.euid != $uid) goto alert;
    goto update_uid_gid;

  state update_setgid!
    case (.auditd.euid != $uid) goto alert;
    case (.auditd.egid != $gid) goto alert;
    goto update_uid_gid;

  state alert!
    $newuid = .auditd.euid;
    $newgid = .auditd.egid;
    report();

  state end!
  /* all went well */
}
From signatures to recurrence equations

For each program point $q$, define a sequence of natural numbers $q_n$, $n \in \mathbb{N}$, so that:
— if we start one thread at $q$,
— and proceed to read $n$ events
then, after the $n$ events have been read, 
**at most $q_n$ threads are in existence.**

We do not compute $q_n$, rather we **generate the equations** they must satisfy, symbolically.
Recurrence equations

\[ \text{init}_0 = 1 \]
\[ \text{init}_{n+1} = \text{newpid}_n + \text{init}_n \]
Recurrence equations

\[ \text{init}_0 = 1 \]

\[ \text{init}_{n+1} = \text{newpid}_n + \text{init}_n \]

\[ \text{newpid}_n = \text{wait}_1_n + \text{wait}_2_n + \ldots + \text{wait}_5_n \]
Recurrence equations

\[
\text{init}_0 = 1 \\
\text{init}_{n+1} = \text{newpid}_n + \text{init}_n \\
\text{newpid}_n = \text{wait}_1^n + \text{wait}_2^n + \ldots + \text{wait}_5^n
\]
Recurrence equations

\[
\begin{align*}
\text{init}_0 &= 1 \\
\text{init}_{n+1} &= \text{newpid}_n + \text{init}_n \\
\text{newpid}_n &= \text{wait}_1^n + \text{wait}_2^n + \ldots + \text{wait}_5^n \\
\text{usu}_n &= \max(\text{usu}_n, \text{alert}_n) \\
\text{usg}_n &= \max(\text{usu}_n, \text{alert}_n)
\end{align*}
\]
**Recurrence equations: syntax**

- **INPUT:** finite collection of equations of the form
  - \( u_{n+k} = a_1 v_n + a_2 w_n + \ldots \), where each \( a_i > 0 \) integer, \( k \) natural
  - \( u_{n+k} = \max(v_n, w_n, \ldots) \), \( k \) natural number

- plus initial conditions \( u_0 = \text{constant} (\geq 1, \text{natural}) \).

- **Goal:** find asymptotic formulae for every \( u_n, v_n, w_n, \text{etc.} \).
Converting to graphs

- Encode those equations as graphs, where:
  - each sequence \((u_n)\) is a vertex \(u\)
  - there are + and max vertices
  - \(u_{n+k} = a_1 v_n + a_2 w_n + ...\) is encoded as
  - \(u_{n+k} = \max(v_n, w_n, ...\) is encoded as

- Close to initial automaton, but not quite the same

- Our algorithm works entirely on the graph.
Converting to graphs

* Encode those equations as graphs, where:
  - each sequence $(u_n)$ is a vertex $u$
  - there are + and max vertices
  - $u_{n+k} = a_1 v_n + a_2 w_n + \ldots$ is encoded as
  - $u_{n+k} = \text{max}(v_n, w_n, \ldots)$ is encoded as

* Close to initial automaton, but not quite the same

* Our algorithm works entirely on the graph.
Converting to graphs

- Encode those equations as graphs, where:
  - each sequence \((u_n)\) is a vertex \(u\)
  - there are + and max vertices
  - \(u_{n+k} = a_1 v_n + a_2 w_n + \ldots\) is encoded as
  - \(u_{n+k} = \text{max}(v_n, w_n, \ldots)\) is encoded as

- Close to initial automaton, but not quite the same

- Our algorithm works entirely on the graph.
Exponential behaviors

Here is the simplest example of a sequence that is asymptotically exponential.

In general, we wish to find a simple criterion to detect which vertices have exponential behaviors (the bad* vertices).

Will depend on the strongly connected components (SCCs) of the graph.

Equation: $u_{n+1}=2u_n$
SCCs (a classic of graph theory)

- A set of vertices $A$ is strongly connected iff every vertex of $A$ is reachable from every other.
- An SCC is a maximal strongly connected set.
- SCCs can be computed in linear time by Tarjan’s algorithm [J.Computing’72]
- The SCCs form the vertices of the so-called condensation graph, which is acyclic
In general, call a vertex \( u \) bad iff

— \( u \) is a + vertex
— the sum of the labels of edges out of \( u \) but remaining in the same SCC is \( \geq 2 \)

A vertex is bad* iff there is a path from that vertex to a bad vertex

Prop. The bad* vertices are exactly those that have exponential behavior.

\[ u \text{ is bad iff } a_1 + a_2 \geq 2 \]
\( (a_3 \text{ is not counted}) \)
Now assume there is no bad vertex. We shall map each SCC $A$ to a number $k$ so that $u_n=O(n^k)$ for every vertex $u$ in $A$.

Look at the trivial SCCs: those with no edge entirely inside the SCC.

$c_{n+1}=5t_n$: if $t_n=O(n^k)$ then $c_n=O(n^k)$ — no increase in degree.
Now assume there is no bad vertex. In every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.
Now assume there is no bad vertex. In every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.
Now assume there is no bad vertex. In every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.

For a non-trivial SCC $A$, an edge $u \rightarrow v$ going out of $A$ is:
- **expensive** iff $u$ is a + vertex
- **cheap** iff $u$ is a max vertex

(Call cheap any edge out of a trivial SCC, too.)
Now assume there is no bad vertex. In every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.

For a non-trivial SCC $A$, an edge $u \to v$ going out of $A$ is:
- **expensive** iff $u$ is a + vertex
- **cheap** iff $u$ is a max vertex

(Call cheap any edge out of a trivial SCC, too.)
Polynomial behaviors 2

- Now assume there is no bad vertex. In every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.

- For a non-trivial SCC $A$, an edge $u \to v$ going out of $A$ is:
  - expensive iff $u$ is a + vertex
  - cheap iff $u$ is a max vertex

- (Call cheap any edge out of a trivial SCC, too.)
Now assume there is no bad vertex. In every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.

For a non-trivial SCC $A$, an edge $u \rightarrow v$ going out of $A$ is:
- **expensive** iff $u$ is a + vertex
- **cheap** iff $u$ is a max vertex

(Call cheap any edge out of a trivial SCC, too.)
Now assume there is no bad vertex. In every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.

For a non-trivial SCC \( A \), an edge \( u \rightarrow v \) going out of \( A \) is:
- **expensive** iff \( u \) is a + vertex
- **cheap** iff \( u \) is a max vertex

(Call cheap any edge out of a trivial SCC, too.)
Now assume there is no bad vertex. In every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.

For a non-trivial SCC $A$, an edge $u \to v$ going out of $A$ is:
- **expensive** iff $u$ is a + vertex
- **cheap** iff $u$ is a max vertex

(Call cheap any edge out of a trivial SCC, too.)
Say that an SCC $A$ has degree $k$ iff every vertex of $A$ is $O(n^k)$.

**Prop.** Assume:
- $A \rightarrow B_i$ (SCCs) through at least one expensive edge, $i=1...m$,
- $A \rightarrow C_j$ (SCCs) through cheap edges only, $j=1...n$.
Assume each $B_i$ has degree $b_i$, each $C_j$ has degree $c_j$.
Then $A$ has degree: $\max \ (\{b_i + 1 | i=1...m\} \cup \{c_j | j=1...n\})$.

I.e., expensive edges increase degree by 1, cheap edges don’t.
Polynomial behaviors: the key case

* Recall that in every SCC, every + vertex has at most one outgoing edge that remains in the SCC, and it has label 1.

* In the example to the right: there is a unique cycle inside the SCC.

* From the equations, \( u_{n+3} = q_{n+2} + u_n \).
  So \( u_{3n+a+1} = q_{3n+a} + q_{3(n-1)+a} + q_{3(n-3)+a} + \ldots + q_a \).
  Let \( q'_n = q_{3n+a} \).
  If \( q_n = O(n^k) \), then \( q'_n = O(n^k) \),
  and therefore \( q'_n + q'_{n-1} + \ldots + q'_0 = O(n^{k+1}) \).
  So \( u_n = O(n^{k+1}) \).  [Increase in degree!

\[ \begin{align*}
  u_{n+1} &= q_n + v_n \\
  v_{n+1} &= w_n \\
  w_{n+1} &= u_n
\end{align*} \]
The algorithm (a bird’s eye view)

- Compute graph, and its SCCs by Tarjan’s algorithm
- Map each SCC to its degree, bottom-up:
  - expensive edges raise degree by 1
  - bad vertices set degree to $+\infty$
- Simple modification of Tarjan’s algorithm, works in linear time
The algorithm (a bird’s eye view)

* Compute graph, and its SCCs by Tarjan’s algorithm

* Map each SCC to its degree, bottom-up:
  — expensive edges raise degree by 1
  — bad vertices set degree to $+\infty$

* Simple modification of Tarjan’s algorithm, works in linear time

\[ \text{degree 0} \] (max of the empty set)
The algorithm (a bird’s eye view)

* Compute graph, and its SCCs by Tarjan’s algorithm
* Map each SCC to its degree, bottom-up:
  — expensive edges raise degree by 1
  — bad vertices set degree to $+\infty$
* Simple modification of Tarjan’s algorithm, works in linear time

---

Fig. 3. Sccs in a graph $G$.

Label at least two, the algo to the first scc. (Vertex labels have been removed for clarity; they are all $+1$, although this is not important.)

Say that a vertex $u$ is bad if and only if some bad vertex is reachable from it, namely if and only if there is a path from $u$ to some bad vertex $v$. We shall see that the bad vertices $u$ are exactly those such that $(u^n)_{n \in N}$ has exponential behavior. In Example 1, $u$ is bad. There is no bad vertex in Example 2 or in Example 4, or in Figure 3.

Proposition 2. For every bad vertex $u$ in $Q$, $(u^n)_{n \in N}$ has exponential behavior.

Proof (Sketch). There is a path from $u$ to some bad vertex $v$. We show that $(v^n)_{n \in N}$ will have exponential behavior, which implies the claim by Lemma 1.

Since $v$ is bad, for some $a \in \{0, 1\}$, $v^n + a$ is defined as a sum of at least two values $s^n$, $t^n$. Since $v$ is again reachable from both $s$ and $t$, $s^n$ and $t^n$ are larger than or equal to $v^n$ for some constant $k$. Hence $v^n + a \geq v^n k$, which entails the claim.

If an scc contains a bad vertex, then all its vertices are bad. Let us consider the case of sccs $A$ without any bad vertex. We shall illustrate the various cases we need to consider on the graph shown in Figure 3. The idea of our algorithm is that we shall iterate on all sccs $A$, from bottom to top, deducing a characteristic degree $d_A$ such that $u^n = \cdot (n d_A)$ for every vertex $u$ in $A$ from the characteristic degrees of sccs below $A$.
The algorithm (a bird’s eye view)

- Compute graph, and its SCCs by Tarjan’s algorithm
- Map each SCC to its degree, bottom-up:
  - expensive edges raise degree by 1
  - bad vertices set degree to $+\infty$
- Simple modification of Tarjan’s algorithm, works in linear time
The algorithm (a bird’s eye view)

- Compute graph, and its SCCs by Tarjan’s algorithm
- Map each SCC to its degree, bottom-up:
  — expensive edges raise degree by 1
  — bad vertices set degree to $+\infty$
- Simple modification of Tarjan’s algorithm, works in linear time
The algorithm (a bird’s eye view)

- Compute graph, and its SCCs by Tarjan’s algorithm
- Map each SCC to its degree, bottom-up:
  - expensive edges raise degree by 1
  - bad vertices set degree to $+\infty$
- Simple modification of Tarjan’s algorithm, works in linear time
The algorithm (a bird’s eye view)

* Compute graph, and its SCCs by Tarjan’s algorithm
* Map each SCC to its degree, bottom-up:
  — expensive edges raise degree by 1
  — bad vertices set degree to $+\infty$
* Simple modification of Tarjan’s algorithm, works in linear time
Implementation(s)

* Two early prototypes in 2013. Integrated into Orchids, 2015.

* 7 signatures are $O(n)$, 1 is $O(n^2)$,
1 is detected as $O(n^3)$ [overapproximated: it is really $O(n^2)$],
1 is $O(1)$.

Rule found to be in $O(n)$ — reason explained on last line
Early implementations (2013)

* One early implementation also gave the coefficient of the dominant monomial. We found later that this is incorrect (see Example 4 in the paper).

* An even earlier implementation compiled the recurrence equations to Sage and to Maple — in very special cases (no max) — very precise result, but becomes unreadable when output is large.
Conclusion

- A very useful tool for warning signature writers of rules that would take up too many resources... which would allow denial-of-service attacks on Orchids itself (!).

- Beyond that, a very general technique for finding asymptotics of certain recurrence equations. Could be used as in [Assaf, PhD, 2015] to deduce quantitative information flow guarantees from an estimation of the number of paths in execution trees?

- Other questions?