Log Auditing through Model-Checking*

Muriel Roger[1,2]
[1]*GIE Dyade, INRIA Rocquencourt*
Domaine de Voluceau B.P. 105
78153 Le Chesnay Cedex, France
`roger@lsv.ens-cachan.fr`

Jean Goubault-Larrecq[1,2]
[2]*LSV, ENS Cachan*
61, av. du président-Wilson
94235 Cachan Cedex, France
`goubault@lsv.ens-cachan.fr`

Abstract

Log auditing is a basic intrusion detection mechanism, whereby attacks are detected by uncovering matches of sequences of events against signatures. We argue that this is naturally expressed as a model-checking problem against linear Kripke models. A variant of the classic linear time temporal logic of Manna and Pnueli with first-order variables is first investigated in this framework. But this logic is in dire need of refinement, as far as expressiveness and efficiency are concerned. We therefore propose a second, less standard logic consisting of flat, Wolper-style linear-time formulae. We describe an efficient on-line algorithm, making the approach attractive for complex log auditing tasks. We also present a few optimizations that the use of a formal semantics affords us.

1 Introduction

Keeping and managing event logs is a standard and fairly universal way of ensuring basic security, whether at the application, system or network level. In particular, it is a cornerstone of *intrusion detection*, which relies on extracting useful information on potential or actual intruders to react accordingly.

Analyzing logs, however, is hard. Detecting intrusion patterns by hand quickly becomes infeasible as logs grow. Most intrusion detection systems include filtering and counting mechanisms [14, 17], but this is not enough in general to eliminate false positives, and new mechanisms that attempt to detect combinations of patterns throughout the logs are required. To take an example from [13], assume we wish to detect an intruder exploiting an old `sendmail` bug on Unix: in this attack, an intruder copies some shell to `/usr/spool/mail/root` at a time where the latter does not exist, sets its setuid bit[1], then sends a fake e-mail message to `root`. On old implementations, when `root` thereafter attempted to read his mail, the ownership of `/usr/spool/mail/root` was simply switched to `root`, making a setuid-bit copy of a shell available to the intruder. Assume these events are logged. Detecting copies of shell files is a good clue that this attack is attempted, and detecting that a non-`root` user is changing setuid bits too, however as a systems administrator we would like to be warned—automatically, if possible—only when the *same* user does both. Reports of one action without the other are *false positives*, where we are warned against a non-existent attack. We may also want to require that an e-mail was indeed sent to root after these two events happened. So we are looking at correlations between different entries in the log—the user has to be the same in each of the copy and setuid events—, together with constraints on the order in which events occur in the log.

Our point is that *signatures*, i.e., specifications of attack patterns, are best expressed in a logic including temporal connectives to express ordering of events. This allows one to describe attacks in a declarative way, free of implementation decisions. As in programming languages, using a *declarative* language allows one to focus on *what* to monitor instead of *how* to monitor. This caters for easier writing and easier understanding of signatures, and improves maintainability of signature files.

Checking a log against temporal logic signatures will then be a *model-checking* problem. While model-checking may have high complexity, we shall pay special attention to efficiency. In particular, we shall see that relying on a logical language with a well-defined

*This work was done as part of Dyade, a common venture between Bull S.A. and INRIA.

[1]In Unix, executables normally run with the rights of the user who launched them. However, files with the setuid bit set will run under the identity of their *creator*. In particular, a shell owned by `root` with the setuid bit set will give any user root privilege.

semantics allows the log auditing engine (the model-checker) to benefit from several optimizations. This is entirely automated, hence safe, contrarily to more imperative languages like Russel [13], where defining and applying these optimizations must be done by hand, and is therefore error-prone.

We wish to do both *on-line* and *off-line* log auditing. In the latter, the log auditing engine is given one log file in its entirety (the traditional approach in model-checking). In on-line log auditing, we require the engine to detect which formulae hold, and when, as soon as possible while the log fills in. Although off-line auditing is useful, on-line auditing is even more so, but places more stringent efficiency constraints on the auditing engine. Our model-checking algorithms will work in both settings. In particular, our second algorithm definitely is on-line, in a precise sense (Theorem 3.12).

The plan of the paper is as follows. After reviewing related work, we explore the use of standard linear-time temporal logic with first-order variables for log auditing in Section 2. This will give the reader a flavor of what temporal logic is, and how we can model-check temporal signatures against (on-line) logs efficiently—at least, as efficiently as possible: we shall show that the model-checking problem for this logic is NP-complete. We identify the shortcomings of this logic, and consequently refine our temporal logic in Section 3, improving the previous approach in many respects. Although, technically, we might have presented the logic, semantics and algorithms of Section 3 directly, the logic of Section 2 is simpler, gives the right intuitions, and justifies many choices leading to Section 3. We report on preliminary practical results and conclude in Section 4.

Related Work. There are many different paradigms to intrusion detection, which may be classified in anomaly detection and misuse detection systems. The former are based on normal activity profiles and detect deviations from these profiles. They are well-suited to detect previously unknown attacks, but usually generate many false positives. On the other hand, misuse detection systems are usually signature-based mechanisms that allow one to keep the number of false positives low, but can only detect behaviors obeying known patterns.

Our work lies clearly in the signature-based, misuse detection field. To be more precise, it is based on a *detection language* (taking the terminology of [6]), which will take the form of a temporal logic. There are now a handful of proposals for detection systems and languages, among which NFR and its N-code [16], Emerald and P-BEST [9], IDIOT [4], STATL [6], ASAX and Russel [13]. Temporal logics have the advantage that they are a high-level, compact and mostly readable notation for events occurring as time passes [10].

Amongst those, the systems closest to our work are IDIOT, STATL and ASAX, and we have borrowed some ideas from each. Our algorithms are inspired mostly by the technique of enabled rules lists of Russel [13], which are already known to provide fast detection algorithms in practice. *Efficiency* is indeed one of our concerns.

However, as Mounji notes, Russel is very low level, and would benefit from a higher-level language that would compile to Russel (as in [5], notably). Instead of compiling to Russel, we choose to provide our own low-level auditing engine, because it requires specific data structures that would be hard—although probably not impossible—to code in Russel.

Our high-level language is, as announced, a temporal language, which seems to make it quite different from the state-transition diagrams of STAT (State Transition Analysis [15]) and its variants, and also from the the colored Petri approach advocated in [8] and used in IDIOT [4]. The particular brand of temporal logic that we use in Section 3 are in fact automata, that is, state-transition diagrams. While STATL relies both on events (from an on-line log) and states (snapshots of the system to monitor), our work focuses on events alone. This allows us to do not only on-line but also off-line log auditing. On the other hand, we are not be able to reason on states. In principle, as shown by Mounji [13], it is possible to piggyback a system like ASAX or ours with a static auditing engine, in charge of maintaining a consistent view of the system. While this is easy to do (embed static queries inside the guards of Section 3.1), this would draw us away from our main point—studying log auditing from the angle of model-checking.

One important point in applying model-checking to log auditing is to give a clean *formal semantics* to our signature language. None of the tools and languages above rests on a formal semantics: it is assumed that the users of intrusion detection systems will somehow grasp, by intuition or experiment, how rules match, in which order they are executed, which have precedence over which others, etc. The only exception we are aware of is Russel, which however is of a rather low level.

Having a formal semantics gives us a reference point for understanding signatures and reasoning about them, and has proved crucial already in other domains of computer science, in particular in the design of programming languages. Notably, this paves the way for other tools on top of the language, e.g., optimizers, debuggers, test case generators, etc. This already proved beneficial in our case by allowing our tool to apply non-trivial optimizations in the course of detection (Section 3.3).

On the other hand, model-checking temporal logics is a well-studied topic [10]. As we use variants of linear-time temporal logic, it would be natural to use the standard automata-theoretic model-checking algorithm [21]: given a formula F, build a Büchi automaton $\mathcal{A}_{\neg F}$ recognizing exactly the models of $\neg F$, then check that the synchronized product of $\mathcal{A}_{\neg F}$ with the automaton $\mathcal{M}$ describing the current model is empty. However, this works for *propositional* linear-time temporal logic, while we require a fragment of *first-order* temporal logic, in which case $\mathcal{A}_{\neg F}$ would be an infinite automaton. Our algorithms can be seen as constructing finite portions of this infinite automaton on demand.

2 Linear Temporal Logic for Log Auditing

Consider again the `sendmail` attack of the introduction. Its signature can be described informally by "find some event in the log stating that some user X copied some file to `/usr/spool/mail/root`, followed by some event where X sets its setuid bit through `chown` or `fchown`, followed by some event stating that `/usr/spool/mail/root` changed ownership to `root`". This is a temporal formula, so it seems a good idea to investigate temporal logics to specify such signatures. Since our Kripke models—the logs—are linear sequences of events, the classic linear-time temporal logic of [11] seems a good first choice. Finding whether a given signature matches some events in the log is then just the *model-checking* problem for formulae over linear Kripke structures. (This choice will turn out to be rather imperfect. However it is worth investigating, if only because it is well-known and will serve as a stepping stone to our more complex logic of Section 3.)

A final requirement for our logic is as follows. Look back at the mail example: we want to be able to check that the user X who copies some file to `/usr/spool/mail/root` is the same as the one who later sets its setuid bit. So we need a temporal logic that is not just propositional temporal logic, but has first-order variables. The syntax and semantics of this logic is described in Section 2.1. Our model-checking algorithm for this logic is described in Section 2.2. We conclude by reporting on experience with this logic in Section 2.3. This will serve to refine our approach in Section 3.

2.1 Syntax and Semantics

Linear-time temporal logic formulae F, G, …, are given by the following grammar:

F	$::=$	A	atomic formulae
	$\mid$	$\neg F$	negation ("not")
	$\mid$	$F \wedge F$	conjunction ("and")
	$\mid$	$F \vee F$	disjunction ("or")
	$\mid$	$\Diamond F$	(strict) sometimes

where it only remains to specify what atomic formulae are. The latter denote the basic observations we wish to make on states of Kripke structures, i.e., on logged events. Now logged events are records:

Definition 2.1 (Record) *A* record R *is a finite map from a set* $\mathcal{L}$ *of* labels *to a set of* values*, which we shall take to be strings. The value* $R(\ell)$*, where* $\ell \in \mathcal{L}$*, will also be noted* $R.\ell$ *for convenience. The* domain $\mathrm{dom}\, R$ *is the set of labels for which* $R.\ell$ *is defined.*

We use *record patterns* as atomic formulae to observe events, i.e., records. For example, the record pattern `{id=X, action="creat", object="^/usr/spool/mail/root$"}` matches any record containing a field `id` that matches the value of variable `X`, a field named `action` matching the regular expression `creat` (i.e., containing the string `creat`: we define regular expression matching as finding some substring that is in the language defined by the regular expression, as in [20]), a field named `object` matching `^/usr/spool/mail/root$` (i.e., exactly equal to the string `/usr/spool/mail/root`), and which finally may contain other fields.

Formally, record patterns are lists of *rows* $id{=}pat$. In a row, id is a label and pat is a *field pattern* defined as either a regular expression r (between double quotes), or a variable name X. We assume no two rows in a given record pattern have the same label.

Our Kripke structures are logs:

Definition 2.2 (Log) *A* log S *is a finite or infinite sequence of records. We let* S_i *be the* i*th record in* S*, if defined. The* domain $\mathrm{dom}\, S$ *of* S *is the set of indices* i *for which* S_i *is defined. The* length $|S|$ *of* S *is the sup of all indices* i *such that* S_i *is defined.*

The reason why we allow the log to be infinite is to handle on-line model-checking: in this setting, the actual log is only a finite prefix of a possibly infinite sequence of events, which grows through time.

Assume a regular expression pattern-matching function Reg that takes a string s and a regular expression r and returns true if and only if s matches r. Then:

Definition 2.3 (Kripke Semantics) *An* environment ρ *is any map from variables to string values.*

A field pattern p matches *a string* s *under* ρ *if and only if either* p *is a variable* X *and* $\rho(X) = s$*, or* p *is a regular expression and* $Reg(s, p)$ *is true.*

Define the relation $S, \rho, i \models F$ *on formulae* F *(*$i \in$

$\operatorname{dom} S$), *as:*

F	$S, \rho, i \models F$ *iff:*
$\{id_1{=}p_1, \ldots, id_n{=}p_n\}$	*for all* $k \in \{1, \ldots, n\}$, $S_i.id_k$ *is defined and* p_k *matches* $S_i.id_k$ *under* ρ
$\neg G$	$S, \rho, i \not\models G$
$G_1 \wedge G_2$	$S, \rho, i \models G_1$ *and* $S, \rho, i \models G_2$
$G_1 \vee G_2$	$S, \rho, i \models G_1$ *or* $S, \rho, i \models G_2$
$\Diamond G$	*for some* $j > i, j \in \operatorname{dom} S$: $S, \rho, j \models G$

2.2 Algorithm

Our algorithm takes as input a finite list of temporal formulae $F_1, \ldots, F_m$ and a finite log, and returns which formulae hold at which states of the log and with which environments. Recall that this algorithm must be *on-line*, i.e., it must output a positive match as soon as it can. In particular, it must work in one pass over the log.

The algorithm is inspired from the implementation of Russel [13], and uses two lists of formulae, $Current$ and $Next$. Given that we are currently reading entry S_i of the log S, $Current$ holds all formulae F such that we would like to know whether $S, \rho, i \models F$. This may depend on deciding questions of the form $S, \rho', i+1 \models F'$; e.g. if $F = \Diamond G$ then $S, \rho, i \models F$ if and only if $S, \rho, i+1 \models G$ or $S, \rho, i+1 \models F$. Instead of deciding right away whether the latter hold, we stash such formulae F' into the $Next$ list. When we have dealt with all formulae in $Current$, we advance to entry S_{i+1} in S (provided it exists), move all formulae from $Next$ to $Current$, set $Next$ to the empty list, and start again.

For example, consider the formula F_0:

```
{op="connection", result="fail",
  subject=X}
∧◊{op="connection", result="pass",
  subject=X}
```

which describes the signature where user X (whoever it might be) fails to connect, then later manages to connect. Write F_1 for the first record pattern above, F_2 for the second, so $F_0 = F_1 \wedge \Diamond F_2$. Consider the log:

i	S_i
1	`op="connection", result="fail", subject="Joe"`
2	`op="connection", result="pass", subject="Joe"`
3	`op="exec", result="pass", object="emacs", subject="Joe"`

Here is how our algorithm runs. First $Next$ is initialized to the empty list, $Current$ to the list containing only formula F_0, i.e., $F_1 \wedge \Diamond F_2$, and we read the first record S_1. $F_1 \wedge \Diamond F_2$ holds at state S_1 if and only if both F_1 and $\Diamond F_2$ hold at state S_1, so we remove $F_1 \wedge \Diamond F_2$ from $Current$, and add both F_1 and $\Diamond F_2$ to $Current$. Consider F_1, and remove it from $Current$: using Reg, it is easy to see that F_1 is satisfied at S_1 if and only X has value `"Joe"`. Now consider the other formula in $Current$, $\Diamond F_2$, and remove it. There is no way to decide whether it holds at S_1 by just looking at S_1. Since $S, \rho, i \models \Diamond F_2$ if and only if $i+1 \in \operatorname{dom} S$ and either $S, \rho, i+1 \models F_2$ or $S, \rho, i+1 \models \Diamond F_2$, we add F_2 and $\Diamond F_2$ to $Next$, therefore postponing the examination of F_2 and $\Diamond F_2$ to the next stage. (The careful reader may have noticed that conjunctions and disjunctions are treated alike: whether we encounter $F_1 \wedge F_2$ or $F_1 \vee F_2$ in $Current$, we generate both F_1 and F_2. The difference will be handled through additional data structures, see below.) At this point, $Next$ contains F_2 and $\Diamond F_2$, and $Current$ is empty, so our work on S_1 is complete.

So we move $Next$ to $Current$, empty $Next$, and read S_2. $Current$ contains just F_2 and $\Diamond F_2$, and we try to see which hold at S_2. First take F_2 and remove it from $Current$: it holds exactly when X equals `"Joe"`. So $\Diamond F_2$ holds at S_1 with X equal to `"Joe"`, and since F_1 also holds at S_1 with X equal to the same value, the formula F_0 holds at S_1 with X equal to `"Joe"`. (We call this argument a *deduction chain*.) The algorithm then goes on finding other environments and states at which the input formula F_0 holds, in the same spirit.

The algorithm requires additional structures. First, we need to represent the sets $C_{F,i}$ of environments ρ such that $S, \rho, i \models F$. Without negation, it is enough to use sets $C_{F,i}$ of the form $\{\rho \mid \rho \preceq \rho_0\}$, where ρ_0 is a single *partial environment*, i.e., a finite map from variables to values, and $\rho \preceq \rho_0$ ("ρ extends ρ_0") if and only if $\rho_{|\operatorname{dom} \rho_0} = \rho_0$. This would allow us to represent sets $C_{F,i}$ as a single partial environment ρ_0. With negation, we use additional environments $\rho_1, \ldots, \rho_n$ as *negative constraints*, and represent $C_{F,i}$ as $\rho_0/\rho_1, \ldots, \rho_n \hat{=} \{\rho \mid \rho \preceq \rho_0$ and $\rho \not\preceq \rho_1$ and ... and $\rho \not\preceq \rho_n\}$.

Definition 2.4 (Constraints) *A* constraint C *is either the special symbol* $\perp$, *or a* $(n{+}1)$*-tuple* $\rho_0/\rho_1, \ldots, \rho_n$ *of partial environments;* ρ_0 *is the* positive constraint, *and* $\rho_1, \ldots, \rho_n$ *are the* negative constraints. *When* $n = 0$, *we allow the shorthand* ρ_0.

The environment ρ satisfies C, *and we write* $\rho \models C$, *if and only if* C *is of the form* $\rho_0/\rho_1, \ldots, \rho_n$, *and* $\rho \preceq \rho_0$, *and* $\rho \not\preceq \rho_i$ *for all* i, $1 \leq i \leq n$. C *is* satisfiable *if and only if* $\rho \models C$ *for some environment* ρ.

The satisfiability of constraints is decidable in polynomial time. Indeed, it is easy to see that $\rho_0/\rho_1, \ldots, \rho_n$ is unsatisfiable if and only if there is an i, $1 \leq i \leq n$, such that $\operatorname{dom} \rho_i \subseteq \operatorname{dom} \rho_0$ and for every $x \in \operatorname{dom} \rho_i$, $\rho_i(x) = \rho_0(x)$.

Second, we need to maintain deduction chains. They are organized as an and-or tree, representing deductions such as "$G_1 \wedge G_2$ holds at S_i if both G_1 and G_2 do" (and branching), or "$\Diamond G$ holds at S_i if either G holds at S_{i+1}, or $\Diamond G$ holds at S_{i+1}" (or branching). To abstract away from implementation details, we shall use the paradigm of logic programming, which is related to and-or trees in an essential way [18]. With each formula F occurring in $Current$ at record i, associate a propositional variable $ok_{F,i}$, whose intended meaning is "F holds at S_i". Deduction chains are maintained through the use of *constrained clauses* of the form $ok_{F_0,i_0} \leftarrow ok_{F_1,i_1}, \ldots, ok_{F_m,i_m} \mid C$ meaning "$S, \rho, i_0 \models F_0$ provided $S, \rho, i_k \models F_k$ for every k, $1 \le k \le m$, and $\rho \models C$". When C is the true constraint $\emptyset$ (no negative constraint, empty positive constraint), we omit the $\mid C$ part; we also omit the $\leftarrow$ sign if C is true and $n = 0$. Constrained clauses are generated by *expansion rules* for formulae F at record i as shown in Figure 1.

Definition 2.5 (On-line Model-Checking Algorithm) *Given temporal formulae $F_1, \ldots, F_m$ (the* signature file*), and a log S, the model-checking algorithm operates as in Figure 2. We say that* $Match(\{id_1 = pat_1, \ldots, id_n = pat_n\}, R)$ succeeds *if there is an environment ρ such that $R.id_k$ is defined and pat_k matches $R.id_k$ under ρ for every k, $1 \le k \le n$; then we let it be the partial environment that maps every pat_k that is a variable to $\rho(pat_k)$.*

Constrained clauses are used to handle deduction chains by constrained resolution. Call any clause of the form $ok_{F,i} \leftarrow\mid C$ a *fact*. Given a set P of constrained clauses, the *immediate consequence operator* T_P maps sets of facts to sets of facts by: $T_P(\phi) \hat{=} \phi \cup \{(ok_{F,i} \leftarrow\mid C_0 \wedge C_1 \wedge \ldots \wedge C_n) \mid (ok_{F,i} \leftarrow ok_{F_1,i_1}, \ldots, ok_{F_n,i_n} \mid C_0) \in P, (ok_{F_1,i_1} \leftarrow\mid C_1) \in \phi, \ldots, (ok_{F_n,i_n} \leftarrow\mid C_n) \in \phi\}$. The *conjunction* of two constraints is defined by $C \wedge \bot \hat{=} \bot \wedge C \hat{=} \bot$, and $(\rho_0/\rho_1, \ldots, \rho_n) \wedge (\rho'_0/\rho'_1, \ldots, \rho'_{n'})$ is $(\rho_0 \cup \rho'_0)/\rho_1, \ldots, \rho_n, \rho'_1, \ldots, \rho'_{n'}$ provided ρ_0 and ρ'_0 agree on $\mathrm{dom}\,\rho_0 \cap \mathrm{dom}\,\rho'_0$; otherwise, the conjunction is $\bot$. The *facts deduced from* P are the elements of $\bigcup_{n=0}^{+\infty} T_P^n(\emptyset)$. The algorithm of Definition 2.5 is completed by a reporting procedure that outputs all facts of the form $ok_{F_j,i} \leftarrow\mid C$ where $C \hat{=} \rho_0/\rho_1, \ldots, \rho_n$ is satisfiable, $1 \le j \le m$, $i \in \mathrm{dom}\,S$ ("Attack #j starting at line i with values: ρ_0"). We stress that our use of constraint logic programming is mostly a means of explanation, and that these facts are deduced incrementally, using a variant of and-or trees, as the algorithm runs.

This is correct and complete is in the following sense:

Theorem 2.6 (Termination, Soundness, Completeness) *Assume S is finite, $s \hat{=} |S|$, and $p \hat{=} \sum_{i=1}^{m} |F_i|$, where $|F|$ denotes the size of the formula F. Then the algorithm of Definition 2.5 terminates in time $O(ps)$.*

Let P be the set of clauses generated by the algorithm of Definition 2.5. Then $S, \rho, i \models F_j$ if and only if there is a fact $ok_{F_j,i} \leftarrow\mid C$ deduced from P, with C satisfiable, such that $\rho \models C$.

Proof. Termination. Take $Current$ and $Next$ to be sets, represented as bit vectors of p bit entries. Indeed, $Current$ and $Next$ will only contain subformulae of $F_1, \ldots, F_m$. Represent $Clauses$ as a list. The set-theoretic operations then take constant time, $Match$ takes time linear in the size of the current record times the size of the matched record pattern. We may assume that we deal with entries from $Current$ in order of decreasing sizes in the loop 4–12: number bit entries corresponding to subformulae in such a way that smaller subformulae get lesser numbers, and start from the highest bits in $Current$. Then each formula in $Current$ will be dealt with exactly once, and there are at most p of them. So the algorithm runs in time $O(ps)$.

Only if. Let us first show that whenever F is a formula that is in $Current$ such that $S, \rho, i \models F$, then there is a fact $ok_{F,i} \leftarrow\mid C$ deduced from P such that $\rho \models C$. This is by induction on $(s - i) + |F|$. As the argument is repetitive, let us deal with the case $F = \Diamond G \in Current$. Since $S, \rho, i \models F$, for some $i' > i$, $i' \le s$, it holds $S, \rho, i' \models G$. In particular $i + 1 \le s$, so $i + 1 \in \mathrm{dom}\,S$, and $S, \rho, i + 1 \models F$ or $S, \rho, i + 1 \models G$. So at the next turn of the loop 2–13, by induction hypothesis some fact of the form $ok_{F,i+1} \leftarrow\mid C$ or $ok_{G,i+1} \leftarrow\mid C$ with $\rho \models C$ is deduced from P. In the first case, the clause $ok_{F,i} \leftarrow\mid ok_{F,i+1}$ allows us to deduce $ok_{F,i} \leftarrow\mid C$; in the second case, use the clause $ok_{F,i} \leftarrow\mid ok_{G,i+1}$. The conjunctive cases $F = G_1 \wedge G_2$, $F = \neg(G_1 \vee G_2)$, $F = \neg\Diamond G$ are dealt with by noticing that if ρ satisfies two constraints C_1 and C_2, then ρ satisfies $C_1 \wedge C_2$.

Observe that F_j occurs in $Current$ at turn i of the loop 2–13, because of line 3. It follows that if $S, \rho, i \models F_j$, then there is a fact $ok_{F_j,i} \leftarrow\mid C$ deduced from P, with C satisfiable, such that $\rho \models C$.

If. Let $ok_{F,i} \leftarrow\mid C$ be any fact deduced from P. Let n be the smallest integer such that it belongs to $T_P^n(\emptyset)$. An easy induction on n shows that for every $\rho \models C$, it holds $S, \rho, i \models F$. In the conjunctive cases, notice that if ρ satisfies $C_1 \wedge C_2$, then ρ satisfies both C_1 and C_2. ☐

While generating constraints is fast, solving them is hard. In fact model-checking is hard:

Theorem 2.7 (NP-Completeness) *Model-checking linear-time temporal logic with first-order variables against finite linear models, that is, deciding whether for some ρ and i, j, it holds $S, \rho, i \models F_j$, where S is*

F	$curr_i(F)$	$next_i(F)$	$ded_i(F)$
$G_1 \wedge G_2$	G_1, G_2	$\emptyset$	$ok_{F,i} \leftarrow ok_{G_1,i}, ok_{G_2,i}$
$G_1 \vee G_2$	G_1, G_2	$\emptyset$	$ok_{F,i} \leftarrow ok_{G_1,i} \quad ok_{F,i} \leftarrow ok_{G_2,i}$
$\Diamond G$ $(i+1 \in \mathrm{dom}\, S)$	$\emptyset$	G, F	$ok_{F,i} \leftarrow ok_{G,i+1} \quad ok_{F,i} \leftarrow ok_{F,i+1}$
$\Diamond G$ $(i+1 \notin \mathrm{dom}\, S)$	$\emptyset$	$\emptyset$	$\emptyset$
$\neg(G_1 \wedge G_2)$	$\neg G_1, \neg G_2$	$\emptyset$	$ok_{F,i} \leftarrow ok_{\neg G_1,i} \quad ok_{F,i} \leftarrow ok_{\neg G_2,i}$
$\neg(G_1 \vee G_2)$	$\neg G_1, \neg G_2$	$\emptyset$	$ok_{F,i} \leftarrow ok_{\neg G_1,i}, ok_{\neg G_2,i}$
$\neg(\Diamond G)$ $(i+1 \in \mathrm{dom}\, S)$	$\emptyset$	$\neg G, F$	$ok_{F,i} \leftarrow ok_{\neg G,i+1}, ok_{F,i+1}$
$\neg(\Diamond G)$ $(i+1 \notin \mathrm{dom}\, S)$	$\emptyset$	$\emptyset$	$ok_{F,i}$

Figure 1. Expansion rules

$Current := \emptyset; Next := \emptyset; Clauses := \emptyset; i := 1;$
while $(i \in \mathrm{dom}\, S)$ {
  $Current := Current \cup \{F_1, \ldots, F_m\};$
  while $(Current \neq \emptyset)$ {
    pick F from $Current$; $Current := Current \setminus \{F\};$
    if F is a record pattern
     then { **if** $\rho_0 \hat{=} Match(F, S_i)$ succeeds **then** $Clauses := Clauses \cup \{ok_{F,i} \leftarrow \mid \rho_0\};$ }
    else if F is a negated record pattern $\neg G$
      then if $\rho_1 \hat{=} Match(G, S_i)$ succeeds **then** $Clauses := Clauses \cup \{ok_{F,i} \leftarrow \mid \emptyset/\rho_1\};$
        else $Clauses := Clauses \cup \{ok_{F,i}\};$
    else { $Current := Current \cup curr_i(F)$;(* see Figure 1. *)
      $Next := Next \cup next_i(F); Clauses := Clauses \cup ded_i(F);$ } }
  $i := i + 1; Current := Next; Next := \emptyset;$ }

Figure 2. On-line model-checking algorithm

finite, is NP-complete.

Proof. Note first that, in the propositional case (no first-order variables), P is a set of ordinary propositional Horn clauses, for which deduced facts (the least model) can be generated in linear time [12]. So, without first-order variables, this problem is polynomial-time.

To show that the problem is in NP, guess the restriction of ρ to the free variables of $F_1, \ldots, F_m$. This is clearly of polynomial size. Checking that $S, \rho, i \models F_j$ for some i, j can be done by running the algorithm of Definition 2.5, replacing the generation of constrained facts $ok_{F,i} \leftarrow\mid C$ either by the generation of the fact $ok_{F,i}$, if $\rho \models C$, or by doing nothing if $\rho \not\models C$. As this algorithm runs in polynomial time, and solving the resulting set of propositional clauses plus the goal clauses $\neg ok_{F_j,i}$ is linear-time, the problem is in NP.

The problem is NP-hard, by reduction from 3-SAT [7], the problem of the satisfiability of sets of clauses with 3 literals. Let $\mathcal{S} \hat{=} \{C_1, \ldots, C_n\}$ be a set of clauses with 3 literals, and $A_1, \ldots, A_k$ be the variables occurring in them. Let S be the log with two records, $S_1 \hat{=}$`{a="true"}` and $S_2 \hat{=}$`{a="false"}`. Produce the signature with one formula `{a="true"}`$\wedge F_1 \wedge \ldots \wedge F_n$, where F_j is obtained from C_j by replacing negative literals $\neg A_i$ by $\Diamond$`{a=`X_i`}` and positive literals A_i by `{a=`X_i`}`. It is clear that $\mathcal{S}$ is satisfiable if and only if the signature matches (at line 1). ☐

Note that this is however much better than the complexity of model-checking linear-time temporal logic against general finite models [19], which is PSPACE-complete. We let the interested reader observe that the model-checking problem is again in NP if we add $\bigcirc$, $\mathcal{U}$, and $\mathcal{W}$ operators; and that it is NP-hard as soon as we have got at least one of $\Diamond$, or $\bigcirc$, or $\mathcal{U}$, or $\mathcal{W}$ and negation.

2.3 Lessons Learnt

Although an NP upper bound is much better in theory than a PSPACE one, in practice this might be terribly inefficient: experimental evaluation is called for. The first author implemented the algorithm of Definition 2.5, which was then improved by Xiaobo Li (Dyade). While it works well in practice, it is fairly slow, but this is not due to NP-completeness. The main problem is the way *repeated events* are dealt with. Assume you wish to detect matches for some temporal formula of the form $A \wedge \Diamond A$, where A is some record pattern, and assume records $S_1, S_{101}, S_{201}, S_{301}, \ldots$, match A with the same values of variables. Then our algorithm will report *all* matches: $A \wedge \Diamond A$ matches at line 1 with the second A matching line 101, also at line 1 with the second A matching line 201, or matching line 301, ..., also at line 101 with the second A matching line 201, or 301, etc. Most of these matches are redundant, and we would prefer some form of *shortest match* reporting, where only $1/101$, $201/301$, possibly also $101/201$ but not $1/201$ or $1/301$ are reported. It is not enough in log auditing to just report on *existence* of matches, but reporting on all matches is too much.

The second lesson learnt is that our choice of the logic of Section 2.1, which is justified by it being standard, is not totally suited to the task of log auditing. In fact, it is both too expressive and not expressive enough.

It is *too expressive*: formulae such as those used in the proof of Theorem 2.7 involving Boolean combinations of modal formulae, of the form $\Diamond F$, are rarely necessary in practice. They are the source of the complication represented by the generation of deduction chain clauses in the algorithm of Definition 2.5, which in turn weigh heavily on the memory and time requirements of the algorithm in practice. In fact, most of the formulae we need are *flat*, namely of the form $(F_1 \wedge \Diamond G_1) \vee \ldots \vee (F_n \wedge \Diamond G_n)$, where $F_1, \ldots, F_n$ are *present formulae*, i.e. formulae whose validity can be decided without having to look at future records, and $G_1, \ldots, G_n$ are themselves flat. We shall use the idea of flat formulae to dispense with clauses in Section 3.

Most of the formulae we need don't use negations either; in fact, the astute reader will have noticed that negations have their share of problems, in that the algorithm is not complete on *infinite* logs in the presence of negation; for example, it never reports that $\neg\Diamond\neg G$ holds at S_1 in an infinite log S where G always holds. In fact, we can only deal with eventuality properties in a complete way: we only have access to a finite prefix of the Kripke model, contrarily to standard model-checking where the whole model is available, in the form of a finite-state automaton.

It is *not expressive enough*: first, our formulae cannot count, which is a shame, since counting is one of the basic mechanisms available in every run-of-the-mill log auditing system. That is, we may require to report a match when F occurs at least k times, giving signature $F \wedge \Diamond(F \wedge \Diamond(F \wedge \ldots \wedge \Diamond F))$ (with k occurrences of F) to the algorithm, but this is awkward, inefficient (because of the repeated events problem mentioned above), and the number k has to be given foremost—the algorithm cannot compute how many times F occurred. Second, it cannot express *parity conditions*: consider the mail attack of the introduction, where the intruder copies some shell to `/usr/spool/mail/root`, sets the setuid bit on it, then sends a fake e-mail message to `root`. We might instead track the setuid bit, so that if the setuid bit is cleared before `root` receives the message, no alert is reported; however, if the setuid bit is set again, we should again consider reporting an alert; and so on. (This is the attack that Mounji considers in [13], p.74.) This cannot be expressed in regular linear time temporal logic, but it can in an extended logic with new temporal operators described by finite automata atop ordinary operators *à la* Wolper [22]. This is what we shall do in the sequel; this will also enable us to do counting easily.

3 Refining the Logic for Log Auditing

Let us design our new logic. First, start with Wolper-style temporal operators. Our particular variant is built from finite automata $\mathcal{A}$ with transitions labeled by formulae, as follows: the operator $W_s^{\mathcal{A}}$, where s is any state of $\mathcal{A}$, is such that $W_s^{\mathcal{A}} F$ holds at state (record) S_i if and only if either s is a final state of $\mathcal{A}$ and F holds at S_i, or there is a transition $s \xrightarrow{G} s'$ in $\mathcal{A}$ such that G holds at S_i and $W_{s'}^{\mathcal{A}} F$ holds at some S_j, $j > i$.

Our Wolper-style formulae differ from those of Wolper [22] in at least two respects. First, the transitions of $\mathcal{A}$ are not labeled by actions of the Kripke model, rather by formulae. (Remember that the only action in our linear Kripke models is $i \mapsto i + 1$.) Second, going from one state s to a successor state s' means finding some later record S_j, $j > i$, where $W_{s'}^{\mathcal{A}} F$ holds, not finding a successor record S_{i+1} where it would hold. This parallels the fact that we don't need the $\bigcirc$ operator in log auditing, only the strict $\Diamond$ operator.

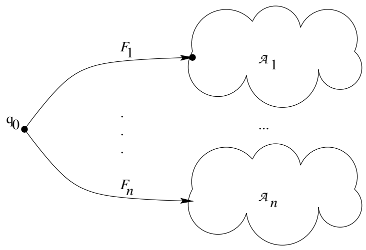
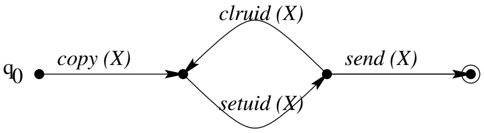
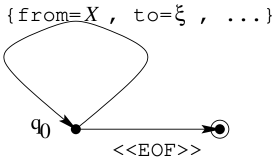


Figure 3. $(F_1 \wedge \Diamond G_1) \vee \ldots \vee (F_n \wedge \Diamond G_n)$

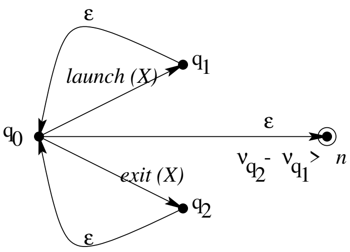
As a first example, note that formulae of the form $(F_1 \wedge \Diamond G_1) \vee \ldots \vee (F_n \wedge \Diamond G_n)$, in the notation of Section 2, correspond to automata of the form shown in Figure 3, where q_0 is the initial state, and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are automata corresponding to $G_1, \ldots, G_n$ respectively. Motivated by the discussion of Section 2.3, we shall restrict formulae that label transitions to be *present formulae*, so that $F_1, \ldots, F_n$ above do not involve any nested



(a) A signature for the mail attack with setuid/clruid alternations



(b) Detecting probing attacks



(c) Tracking launch/exit events

Figure 4. A few Wolper-style formulae

automata constructions. In fact, we shall even restrict the label formulae F further, so that the set of environments ρ such that $S, \rho, i \models F$ is representable in a simple way, as just a positive constraint ρ_0; this will simplify the design of the model-checker, by disallowing negations and disjunctions from label formulae. Note that disallowing disjunctions in F_j does not significantly restrict the expressive power of the logic, since labelling a transition of $\mathcal{A}$ by $F_1 \vee F_2$ can be simulated by having two transitions, one labeled F_1, the other F_2.

Wolper-style formulae also allow us to encode loops in formulae. For example, the refined mail attack of Section 2.3 ([13], p.74), can be specified by the formula $W_{q_0}^{\mathcal{A}} true$, where $\mathcal{A}$ is shown in Figure 4(a); we take $copy(X)$, $setuid(X)$, $clruid(X)$, $send(X)$ to denote record patterns recognizing copy, setuid, clruid, and mail sending events done by X.

In the presence of loops, it becomes important to distinguish between two kinds of first-order formulae: *rigid* variables $X, Y, \ldots$, can only assume a fixed value throughout the sequence of records matched by a given signature—these were the only kind of variables in Section 2—while *flexible* variables [11] ξ, η, $\ldots$, may assume values that vary from record to record. To illustrate why this is needed, consider a signature for detecting a form of probing attack against a firewall, where the attacker assumes a unique identity X and tries to connect to several machines behind the firewall. We might specify this in the logic of Section 2 by writing, say:

$$\texttt{\{from=}X\texttt{, to=}X_1\texttt{\}} \wedge \Diamond(\texttt{\{from=}X\texttt{, to=}X_2\texttt{\}} \wedge \Diamond(\texttt{\{from=}X\texttt{, to=}X_3\texttt{\}}))$$

to detect cases where X tries to connect to three (possibly identical) machines X_1, X_2, and X_3. A better solution, using Wolper-style operators, is the formula $W_{q_0}^{\mathcal{A}} true$ where $\mathcal{A}$ is shown in Figure 4(b); the final state is circled, and `<<EOF>>` is a new record pattern that only matches the end of file. While X has to be rigid, so that we detect attacks where the same `from` field is present, we definitely do not want the `to` field to be the same, whence the use of the flexible variable ξ. (We might just omit the $\texttt{to} = \xi$ row here; flexible variables will be useful in guards, see below; also for reporting the sequence of their values along matching runs.)

Model-checking a formula $W_s^{\mathcal{A}} F$ works by traveling along transitions in the automaton $\mathcal{A}$ (while advancing in the log), until a final state is reached, where checking F starts. *Counting* can now be achieved by a simple trick. With each state s of $\mathcal{A}$, create a flexible variable ν_s that counts the number of times we go through state s in $\mathcal{A}$. Then, in the example of Figure 4(b), we may report ν_{q_0} on reaching the end of the log: for each match, the value of $\nu_{q_0} - 1$ at the end of the log will be the number of times X tried to connect to a machine ξ, for each X.

This form of counting can also be used to reach other goals. In particular, consider that you wish to detect situations where a user X manages to launch more than n copies of an application whose license only allows for n distinct active copies. Assume that no copy is launched at the start of the log, and we track events $launch(X)$ (application launch) and $exit(X)$ (application exit) from this point on. Then $W_{q_0}^{\mathcal{A}} true$, where $\mathcal{A}$ is the automaton $\mathcal{A}$ of Figure 4(c), does the trick.

This example shows a few new constructions. First, the transition from q_1 to q_0, and the one from q_2 to q_0, are *ϵ-transitions*: once control reaches q_1 (resp. q_2), it may spontaneously jump to q_0, without advancing in the log—contrarily to non-ϵ transitions, which can only be traversed by advancing in the log. The transition from q_0 to the final state is also an ϵ transition, but it is subscripted with a *guard* $\nu_{q_2} - \nu_{q_1} > n$. (No guard means $true$.) The meaning of this is that the signature only matches provided control went through state q_2 more than n times more than through state q_1, i.e., that the number of active copies of the application is greater than n, a violation of the license.

Guards are given in a side-effect free language including equality, inequality and pattern-matching tests, plus a few other constructions. We won't describe them in detail, as this is secondary. Similarly, the actual concrete syntax we use for describing Wolper-style formula is in fact a programming language syntax that compiles to automata just like programs can be translated to control-flow graphs [1]. The actual concrete syntax is also not essential to this paper, and we won't describe it.

3.1 The Logic

We are now ready to define the syntax and semantics of the logic. Since, in fact, our only formulae will be Wolper-style formulae $W_s^{\mathcal{A}}F$, F will also be a Wolper-style formula, and we may simplify the presentation by taking formulae of the restricted form $W_s^{\mathcal{A}}true$. First, we define the present formulae that we use to label transitions of our automata:

Definition 3.1 (Present Definite Formulae) *Let $\mathcal{V}_r$, $\mathcal{V}_f$, and $\mathcal{V}_c$ be pairwise disjoint sets of so-called* rigid*,* flexible*, and* counter *variables. A* record pattern *A is a list of* rows *id=pat, where id is a label and pat is a* field pattern*, defined as either a regular expression r, a rigid variable X, or a flexible variable ξ. We assume no two rows in a given record pattern have the same label.*

The set $\mathcal{P}$ of present definite *formulae P, Q, ..., consists of* record patterns *A, plus the special end-of-file declaration* `<<EOF>>`.

We assume that finite logs S end with a special *end-of-file* record, and that `<<EOF>>` only matches this end-of-file record. This does not restrict the generality of our approach, and allows `<<EOF>>` to be matched at all.

Note that counter variables (the ν variables) are not allowed in record patterns. We extend the $Match$ function of Definition 2.5 to deal with the case of `<<EOF>>`. It is easy to see that $Match$ still succeeds, returning a unique partial environment ρ, or fails, deterministically. It is convenient here to define partial environments as triples (ρ_r, ρ_f, ρ_c) mapping rigid, flexible and counter variables respectively to their values. Because values were assumed to be strings, but counters are integers, we also tacitly assume the use of conversion functions between each data type, à la Perl [2]. Other solutions are possible, e.g., defining an algebra of data types; this is however out of the scope of this paper.

Assume a language $\mathcal{G}$ of *guards* built on rigid, flexible, and counter variables. As announced earlier, we leave the language $\mathcal{G}$ unspecified. Its semantics is given by a total computable *evaluation function* $[\![\_]\!]\_$ such that for any guard $g \in \mathcal{G}$ and any partial environment $\rho \hat{=} (\rho_r, \rho_f, \rho_c)$, $[\![g]\!]\rho$ is a Boolean value. If g labels a transition $q \xrightarrow[g]{a} q'$, $[\![g]\!]\rho$ will be used to decide whether we can go from q to q'.

Definition 3.2 (Wolper-Style Formulae) *Let ϵ be a special symbol, outside of $\mathcal{P}$. A* Wolper-style formula *F is a tuple (Q, I, T, Δ), where $Q \subseteq \mathcal{V}_c$ is a finite set of so-called* control states*, $I \in Q$ is the* initial state*, $T \subseteq S$ is the set of* final states*, and $\Delta \subseteq S \times (\mathcal{P} \cup \{\epsilon\}) \times \mathcal{G} \times S$ is the* transition relation*.*

Note that every control state is identified with a counter variable: this trick allows us to naturally map each state q to a counter variable ν_q (i.e., $\nu_q = q$); in implementations, we may of course choose different data structures for states and variables. We write $q \xrightarrow[g]{a} q'$ when (q, a, g, q') is in Δ. In the sequel, we restrict Wolper-style formulae F to those obeying the following assumptions 3.3, 3.4, and 3.5. This does not restrict the generality of the approach, but failing to make these assumptions would result in definite complications. (The ϵ-determinism Assumption 3.5 is needed for technical reasons in the proof of Theorem 3.11.)

Assumption 3.3 *F has no cycle of ϵ-transitions, i.e., it does not contain any sequence of transitions $q_1 \xrightarrow[g_1]{\epsilon} q_2 \xrightarrow[g_2]{\epsilon} \dots \xrightarrow[g_n]{\epsilon} q_1$.*

Assumption 3.4 *There is no sequence of ϵ-transitions from the initial state to any final state in F.*

Assumption 3.5 *For every two distinct ϵ-transitions $q \xrightarrow[g_1]{\epsilon} q_1$ and $q \xrightarrow[g_2]{\epsilon} q_2$ with the same start state q, g_1 and g_2 cannot both hold, i.e., for no partial environment ρ it is the case that $[\![g_1]\!]\rho = true$ and $[\![g_2]\!]\rho = true$.*

To define the semantics of formulae, first define what it means to *enrich* a partial environment, e.g., by the result of a $Match$ operation. On rigid variables, enriching ρ_r by ρ'_r succeeds if and only if ρ_r and ρ'_r agree on $\operatorname{dom} \rho_r \cap \operatorname{dom} \rho'_r$, and yields $\rho_r \cup \rho'_r$, the partial environment mapping every rigid variable X in $\operatorname{dom} \rho_r$ to $\rho_r(X)$, and every rigid variable X in $\operatorname{dom} \rho'_r$ to $\rho'_r(X)$. (Enrichment first checks that the values of rigid variables do not change.) On flexible variables, enriching ρ_f by ρ'_f means computing $\rho_f \oplus \rho'_f$, defined as mapping every $\xi \in \operatorname{dom} \rho'_f$ to $\rho'_f(\xi)$, and every $\xi \in \operatorname{dom} \rho_f \setminus \operatorname{dom} \rho'_f$ to $\rho_f(\xi)$. (New values hide old values.)

Definition 3.6 ($EnrichMatch$) *For every present definite formula P, record R, and partial environment $\rho \hat{=} (\rho_r, \rho_f, \rho_c)$, if $Match(P, R)$ succeeds and returns*

$(\rho'_r, \rho'_f, \{\})$*, and if* ρ_r *and* ρ'_r *agree on* $\mathrm{dom}\,\rho_r \cap \mathrm{dom}\,\rho'_r$*, let* $EnrichMatch(P, R, \rho) \hat{=} (\rho_r \cup \rho'_r, \rho_f \oplus \rho'_f, \rho_c)$*. Otherwise,* $EnrichMatch(P, R, \rho)$ *is undefined.*

Note that $Match$ does not involve counter variables. However, we let $\rho_c[\nu{+}{+}]$ denote the environment that maps ν to $\rho_c(\nu) + 1$, and every other counter variable $\nu' \in \mathrm{dom}\,\rho_c$ to $\rho_c(\nu')$. (This increments the counter variable ν.) By extension, if $\rho = (\rho_r, \rho_f, \rho_c)$, then we let $\rho[\nu{+}{+}]$ be $(\rho_r, \rho_f, \rho_c[\nu{+}{+}])$.

While we might define the semantics of Wolper-style formulae in terms of a relation $S, \rho, i \models F$ as in Definition 2.3, we prefer a more operational definition, where not only the index i of the first record S_i where F matches is explicit, but also all indices where each control state of F is matched. This is useful notably in reporting attacks: the list of records S_i constituting the attack is more useful than the first record alone.

Definition 3.7 (Run) *A* partial run *of a Wolper-style formula* $F \hat{=} (Q, I, T, \Delta)$ *against a log* S *is a finite sequence* $\sigma \hat{=} (\sigma_1, \ldots, \sigma_\ell)$ *(*$\ell \geq 0$*) of tuples* $\sigma_j \hat{=} (q_j, i_j, \rho_j)$*,* $1 \leq j \leq \ell$*, each consisting of a state* q_j *of* Q*, an index* $i_j \in \mathrm{dom}\,S$ *of a record in the log* S*, and a partial environment* ρ_j*, such that, whenever* $\ell \geq 1$*:*

1. $q_1 = I$ *(start condition);*
2. *for every* $1 < j \leq \ell$*, there is a transition* $q_{j-1} \xrightarrow[g]{a} q_j$ *in* Δ *such that:*
   (a) a *is a present definite formula* P*,* $\rho' \hat{=} EnrichMatch(P, S_{i_{j-1}}, \rho_{j-1})$ *is defined,* $\rho_j \hat{=} \rho'[\nu_{q_j}{+}{+}]$*,* $[\![g]\!]\rho_j$ *is true (the guard holds), and* $i_j > i_{j-1}$ *(advancing in the log);*
   (b) *or* $a = \epsilon$*,* $\rho_j = \rho_{j-1}[\nu_{q_j}{+}{+}]$*,* $[\![g]\!]\rho_j$ *is true, and* $i_j = i_{j-1}$ *(not advancing in the log).*

A run*, or a* complete run*, is a run* σ *as above where in addition* $\ell \geq 1$ *and* $q_\ell \in T$*.*

The domain $\mathrm{dom}\,\sigma$ *of* σ *is* $\{i_1 < i_2 < \ldots < i_\ell\}$*. The* span *of* $\{i_1, i_2, \ldots, i_\ell\}$*, and by extension of* σ *is* $\mathrm{span}\,\sigma \hat{=} [i_1, i_\ell]$ *(the empty set if* $\ell = 0$*). The* length $\mathrm{len}\,\sigma$ *is* ℓ*. If* $\ell \geq 1$*, the* beginning *of* σ *is* i_1*, its* end *is* i_ℓ*, its* end state *is* q_ℓ*, and its* end environment $end\rho(\sigma)$ *is* ρ_ℓ*.*

We define the strict orderings $<_{lex}$ *and* $\prec$ *on domains of runs (and, by extension, on runs), as follows: for every domains* $D \hat{=} \{i_1 < i_2 < \ldots < i_\ell\}$ *and* $D' \hat{=} \{i'_1 < i'_2 < \ldots < i'_{\ell'}\}$*,* $D <_{lex} D'$ *if and only if* $i_1 = i'_1$*,* $i_2 = i'_2$*, …,* $i_{j-1} = i'_{j-1}$*,* $i_j < i'_j$ *for some* j*,* $1 \leq j \leq \min(\ell, \ell')$*. We let* $D \prec D'$ *if and only if* $[i_1, i_\ell] \subsetneq [i'_1, i'_{\ell'}]$*, or* $[i_1, i_\ell] = [i'_1, i'_{\ell'}]$ *and* $D <_{lex} D'$*.*

We use the strict ordering $\prec$ to define the notion of shortest run that we were longing for in Section 2.3: a *shortest run* of F against S is one that is minimal with respect to $\prec$. Our reasons for choosing this particular definition are that, first, it seems to give results that user expect in practice—modulo a few changes that we describe below—, and second, it is simple enough to design a model-checking algorithm for this logic that only returns shortest runs. Moreover, shortest runs are optimal in a precise sense (see Theorem 3.8), as far as minimizing redundant information reporting is concerned.

Let us explain why this notion of shortest run seems adequate. First, consider the repeated events problem of Section 2.3, where some record pattern A matches records S_1, S_{101}, S_{201}, S_{301}, $\ldots$, and we look for matches of $A \wedge \Diamond A$, i.e., the Wolper-style formula $q_0 \xrightarrow[true]{A} q_1 \xrightarrow[true]{A} q_2$ with $I \hat{=} q_0$, $T \hat{=} \{q_2\}$. Since any shortest run has minimal span, the only shortest runs here are S_1/S_{101}, S_{101}/S_{201}, S_{201}/S_{301}, etc. Taking runs with shortest spans only also allows us to report a license violation in the example of Figure 4(c) as soon as the number of active copies exceeds n, and only when it does so. Reporting all runs would force us to report at every subsequent record—at least until enough copies have been exited—that the number of active copies still exceeds n, which is clearly spurious information.

Our notion of shortest run, however, does not just rest on minimum spans, but also on a lexicographic ordering of domains. To see why such a solution is needed, consider the case of the mail attack of the introduction, detected by the Wolper-style formula $q_0 \xrightarrow[true]{copy(X)} q_1 \xrightarrow[true]{setuid(X)} q_2 \xrightarrow[true]{send(X)} q_3$, where $I \hat{=} q_0$, $T \hat{=} \{q_3\}$. Consider a log S where S_1 is $copy(a)$ for some a, both S_{11} and S_{20} are $setuid(a)$, and S_{30} is $send(a)$, with no intervening $copy(a)$ or $send(a)$ in between. Then there are at least two runs with minimum span $[1, 30]$, one with domain $\{1, 11, 30\}$, the other with domain $\{1, 20, 30\}$. Our definition of shortest runs selects the former, and discards the latter. The choice of one over the other is arbitrary, but a choice has to be made: reporting both runs is redundant.

More subtly, consider the mail attack with setuid/clruid alternations of Figure 4(a), and assume S_1 is a $copy(a)$ event, S_{20} is a $setuid(a)$ event, S_{30} is a $clruid(a)$ event, S_{40} is again a $setuid(a)$ event, and S_{100} is a $send(a)$ event. Several runs exist for this formula against this log, even with the same span $[1, 100]$: one has domain $\{1, 20, 30, 40, 100\}$, another $\{1, 20, 100\}$, yet another $\{1, 40, 100\}$. Using shortest runs, only the one with domain $\{1, 20, 30, 40, 100\}$, the most informative one, is reported.

The important properties of shortest runs are given in the following Theorem. Part (i) asserts that shortest runs are rare, and are optimal as far as minimizing spurious reports is concerned: although there may be several runs of F against S with beginning i, they can only differ by the values of variables; since attacks are characterized by the sequence of records S_i in the log S that describes them, the shortest run semantics reports at most one attack for each formula F and beginning i. Part (ii) asserts that shortest runs are complete: no information is lost by only reporting shortest runs, instead of all runs.

Theorem 3.8 (Optimality, Completeness) *Let $i \in \operatorname{dom} S$, and F a Wolper-style formula. Then:*
(i) *All shortest runs of F against S with beginning i have the same domain;*
(ii) *If there is a run of F against S with beginning i, then there is a shortest one.*

Proof. First show that $\prec$ is total on domains of runs with beginning i, i.e., for any $D \hat{=} \{i_1 < i_2 < \ldots < i_\ell\}$ and $D' \hat{=} \{i'_1 < i'_2 < \ldots < i'_{\ell'}\}$, with $i_1 = i'_1 = i$, then $D = D'$ or $D \prec D'$ or $D' \prec D$. If $i_\ell < i'_{\ell'}$ then $D \prec D'$, if $i'_{\ell'} < i_\ell$ then $D' \prec D$. So it remains to deal with the case $i_\ell = i'_{\ell'}$, where D and D' have the same span. If there is an index j such that $1 \leq j \leq \min(\ell, \ell')$ and $i_j \neq i'_j$, take j to be smallest, then either $i_j < i'_j$ and therefore $D \prec D'$, or $i_j > i'_j$, so $D' \prec D$. Otherwise, no such j exists, and we claim that $D = D'$. Without loss of generality, assume that $\ell \leq \ell'$. Then the sequence $i_1 < i_2 < \ldots < i_\ell$ is a prefix of $i'_1 < i'_2 < \ldots < i'_{\ell'}$, in particular $i_\ell = i'_\ell$. Since D and D' have the same span, $i_\ell = i'_{\ell'}$, so $i'_\ell = i'_{\ell'}$. If $\ell < \ell'$, this would entail $i'_\ell < i'_{\ell'}$ since the sequence D' is strictly increasing, which would be a contradiction. So $\ell = \ell'$, hence $D = D'$.

So we cannot have two shortest runs of F against S with beginning i, with different domains, as one of the domains should be shorter (w.r.t. $\prec$). This proves (i).

Secondly, we claim that $\prec$ is well-founded on domains of runs. Take any infinite decreasing sequence of domains D_i, $i \geq 0$: there must be an i_0 such that the span of D_i is the same for all $i \geq i_0$. Indeed, the inclusion ordering for spans is well-founded. Without loss of generality, assume that $i_0 = 0$ and the span of every D_i is $[i, i']$. Then $\prec$ is included in the lexicographic product of at most $i' - i + 1$ copies of the $<$ well-founded ordering on integers $\geq i$, which is well-founded.

It follows that, if there is a run of F against S with beginning i, there is a least one for $\prec$, that is, there is a shortest such run. So (ii) holds. $\square$

3.2 Model-Checking

Our model-checking algorithm for the logic is similar to Definition 2.5, except for a few points. First, as desired, we don't need $Clauses$ any longer, or any mechanism for solving deduction chains. Second, while $Current$ and $Next$ used to be lists or sets, but this was essentially unimportant (except that using sets allowed us to derive optimal worst-case complexity bounds in the propositional case), here it will be important that $Current$ and $Next$ are FIFO *queues*: this is how we implement the shortest run semantics of Section 3.1.

Our operations on queues f are: testing whether f is empty, creating a new, empty queue ($f := \texttt{newqueue}()$), pushing an element a at the back of f ($\texttt{enqueue}(a, f)$), and popping an element a from the front of f, if f is not empty ($a := \texttt{dequeue}(f)$). The information we keep on queues $Current$ and $Next$ are tuples $(F, pid, \delta, \sigma, once, retrigger)$, where F is a Wolper-style formula, pid is an integer, δ is a transition $q \xrightarrow[g]{P} q'$ of the automaton F such that P is a present definite formula, σ is a partial run of F against S whose end state is q (if its length is ≥ 1), and $once$ and $retrigger$ are Booleans : $once$ is true if we only want to know whether F holds now (at the current position S_i in the log S), and false if we want to know whether F is true now or later; $retrigger$ is used to implement optimizations, for now simply assume that $retrigger$ is true.

Intuitively, the algorithm of Definition 3.9 simulates the execution of a parallel non-deterministic machine, running non-deterministic *threads* in parallel. For each signature (Wolper-style formula) F, and each $i_0 \in \operatorname{dom} S$, a new thread is launched to test whether there is a run of F against S with beginning i_0, and pid is a unique integer associated with this thread, as in Unix. Contrarily to Unix, however, each thread is a *non-deterministic* machine, which may split at branch points, just like a non-deterministic Turing machine [7]. In particular, the $Current$ and $Next$ queues may contain several tuples with the same pid, corresponding to different branches of the same non-deterministic thread.

The fact that $(F, pid, \delta, \sigma, once, retrigger)$—at least when $once = false$, which we assume for the moment—is in the $Current$ queue, with $\delta = q \xrightarrow[g]{P} q'$, and that we are looking at record S_i in the log S, denotes that thread pid is waiting for some record $S_{i'}$, $i' \geq i$, that would match P, and such that g then evaluates to $true$, to advance to control state q'. Advancing to state q' means following any sequence of ϵ transitions (remember that there are no ϵ-cycles by Assumption 3.3, so this must terminate), until we reach non-ϵ-transitions, which are enqueued onto $Next$. To do so, the model-checker tests right away whether S_i (the case $i' = i$) matches P, whether this makes g true, and if so advances to state q'. It also enqueues $(F, pid, \delta, \sigma, once, retrigger)$ onto $Next$, to check for later records $S_{i'}$, $i' > i$. This is one instance of a non-deterministic split.

This normal behavior is modified when $once$ is $true$. If so, then the model-checker won't enqueue $(F, pid, \delta, \sigma, once, retrigger)$ onto $Next$. This happens in the case when the model-checker tries to find a run of the signature F against S with given beginning i_0. Then it really wants to match P, where P labels one of the first non-ϵ transitions of F, against S_{i_0}, not any later record; so it should set $once$ to $true$.

The purpose of the $retrigger$ Boolean will be explained in Section 3.3. For now, we may notice that $retrigger$ is always $true$ in the following algorithm, and therefore plays no role (yet). The places where we shall modify the algorithm to optimize it are marked with an asterisk, in lines 7 of `advance` and .11 of the main algorithm. The algorithm is as follows, and is parameterized by an attack reporting function `report`; the function `newpid`() creates a fresh integer pid, while () is the empty partial run and $\sigma :: (q, i, \rho)$ denotes the partial run σ with (q, i, ρ) added at the end.

Definition 3.9 (On-line Model-Checking) *Let $Kill$ be a global variable containing finite sets of integers.*

Define the function `advance` *(*advance to state q *in* F *when at record* S_i*, with environment* ρ*, current partial run* σ*, thread* pid*, and Boolean* $once$*): see Figure 5, top. This updates* $Kill$ *and the queue parameter* f*.*

Let $F_1, \ldots, F_m$ *be Wolper-style formulae (the signature file). Let* I_j *be the initial state of* F_j*. Model-checking* $F_1, \ldots, F_m$ *against the log* S *operates as in Figure 5, bottom.*

Lemma 3.10 (Termination) *The algorithm of Definition 3.9 terminates on any finite log* S*. More generally, for any* $i_0 \in \operatorname{dom} S$*,* i *reaches the value* i_0 *in line* .3 *of the algorithm in finite time.*

Proof. The `advance` function terminates because of Assumption 3.3. Recall that the evaluation function $[\![\_]\!]_\_$ is total, i.e., terminating. So the loop body .5–.12 terminates. Since this loop body only enqueues on the $Next$ queue, and dequeues from the $Current$ queue, eventually $Current$ will be empty, hence the loop .4–.12 terminates. The claim follows. □

An invariant of the algorithm of Definition 3.9 is that for every entry $(F, pid, \delta, \sigma, once, retrigger)$ in $Current$ or $Next$, σ is a non-empty partial run of F against S, and $\delta = q \xrightarrow[g]{P} q'$ for q equal to the end state of σ. It follows in particular that every σ that `report` reports at line 4 of `advance` is a valid partial run. Moreover, σ must be a complete run since its end is a final state of F, because of the test at line 3 of `advance`. So only valid runs are reported.

The purpose of $Kill$ is to remember the pids corresponding to pairs (F_j, i_0), such that a run of F_j with beginning i_0 has already been reported. Note that $pids$ are tested in line .6, so that once a run with a given pid has been reported, no entry with the same pid is ever re-enqueued again. As pids are in one-to-one correspondence with pairs (F_j, i_0), this algorithm only reports at most one run for each pair (F_j, i_0).

For every partial run ς, let $\varsigma_{|i}$ denote the maximal prefix of ς of domain included in $[1, i]$. Another invariant of the algorithm is that: $(*)$ if no run of F_j with beginning i_0 has yet been reported, and $i \geq i_0$ at the start of the loop .4–.12, then for every run ς of F_j with beginning i_0, there is an entry $(F, pid, \delta, \sigma, once, retrigger)$ in $Current$ such that either $once = false$, $i > i_0$ and $\sigma = \varsigma_{|i-1}$, or $once = true$, $i = i_0$ and $\sigma = \varsigma_{|i}$. This is indeed true when $i = 1$, since then the call to `advance` at line .3 has enqueued all entries of the form $(F, pid, \delta, \sigma, true, true)$, where σ ranges over all partial runs of the form $((q_1 = I_j, i_0, \{\}), (q_2, i_0, \{\}), \ldots, (q_n, i_0, \{\}))$ where $\delta = q_n \xrightarrow[g]{P} q'$, and certainly any run of F_j with beginning i_0 must start with such a partial run. (Remember that by Assumption 3.4, no sequence of ϵ-transitions leads from the initial state to any final state.) Then we use induction on computations, using the easily proved fact that if $(*)$ holds, then after executing the loop .4–.12, if no run of F_j with beginning i_0 has yet been reported, then for every run ς of F_j with beginning i_0, there is an entry $(F, pid, \delta, \sigma, once, retrigger)$ in $Next$ such that $once = false$ and $\sigma = \varsigma_{|i}$. Indeed, if $i \in \operatorname{dom} \varsigma$, then the corresponding entry is added by the call to `advance` at line .8, otherwise is is added by one of the calls to `enqueue`, at lines .10, .11 or .12.

It follows that for every F_j, for every $i_0 \in \operatorname{dom} S$, at least one run of F_j with beginning i_0 will be reported, if there is one. Therefore:

Theorem 3.11 (Soundness, Completeness) *The algorithm of Definition 3.9 reports exactly one shortest run of* F_j *against* S *with beginning* i_0*, for all* $i_0 \in \operatorname{dom} S$ *such that there is such a run, and all* j*,* $1 \leq j \leq m$*.*

Proof. Taking into account the above considerations, it only remains to show that the unique reported run is shortest. Because records S_i are processed in increasing i order, only runs of F_j with beginning i_0 with smallest span are reported. So it remains to show that amongst those, the reported one is the lexicographically smallest. The (intuitive, but slightly wrong) idea is that for every two entries $(F_j, pid, \delta, \sigma, once, retrigger)$ and $(F_j, pid, \delta', \sigma', once', retrigger')$ in $Current$ at the start of the loop .4–.12, such that the former occurs before the latter in the queue (i.e., it will be dequeued first), then $\operatorname{dom} \sigma <_{lex} \operatorname{dom} \sigma'$. Intuitively, this condition is preserved because dequeued entries in $Current$ are re-enqueued in the same order onto $Next$. The subtle point

1 **function** advance$(F, q, i, \rho, \sigma, pid, once, f)$ {
2 $\rho' := \rho[\nu_q{+}{+}]$; $\sigma' := \sigma :: (q, i, \rho')$;
3 **if** q is final in F
4 **then** { $Kill := Kill \cup \{pid\}$; (* found shortest run, remember to kill others. *) report(F, σ); }
5 **else for every** transition $\delta \hat{=} q \xrightarrow[g]{a} q'$ in F {
6 **if** $q = \epsilon$ **then** { **if** $[\![g]\!]\rho$ **then** advance$(F, q', i, \rho', \sigma', pid, once, f)$; }
7 **else** enqueue$((F, pid, \delta, \sigma', once, true^*), f)$; } }

.1 $Current :=$ newqueue(); $Next :=$ newqueue(); $Kill := \emptyset$; $i := 1$;
.2 **while** $(i \in \mathrm{dom}\, S)$ {
.3 **for** $j = 1..m$ { advance$(F_j, I_j, i, \{\}, (),$ newpid()$, true, Current)$; }
.4 **while** $(Current$ is not empty$)$ {
.5 $(F, pid, \delta \hat{=} (q \xrightarrow[g]{P} q'), \sigma, once, retrigger) :=$ dequeue$(Current)$;
.6 **if** $pid \notin Kill$ **then** {
.7 **if** $\rho' \hat{=} EnrichMatch(P, S_i, end\rho(\sigma))$ is defined
.8 **then** { **if** $[\![g]\!]\rho'$ (* guard is true: advance. *)
.9 **then** { advance$(F, q', i, \rho', \sigma, pid, false, Next)$;
.10 **if** $retrigger$ **and not** $once$ **then** enqueue$((F, pid, \delta, \sigma, false, retrigger), Next)$; }
.11 **else if** $true^*$ **and not** once **then** enqueue$((F, pid, \delta, \sigma, false, retrigger), Next)$; }
.12 **else if not** once **then** enqueue$((F, pid, \delta, \sigma, false, retrigger), Next)$; } }
.13 $i := i + 1$; swap $Current$ and $Next$; }

Figure 5. On-line model-checking

is that additional entries, namely those added by the call to advance at line .9, preserve the condition, since this call is made before we re-enqueue at line .10. However the condition $\mathrm{dom}\, \sigma <_{lex} \mathrm{dom}\, \sigma'$ does not quite work.

Instead, define the orderings $<_i$ on domains $D, D' \subseteq [1, i{-}1]$ by $D <_i D'$ if and only if $D \cup Z <_{lex} D' \cup Z$ for all $Z \subseteq [i, +\infty)$, $Z \neq \emptyset$. For every $k \geq 1$, map the kth entry $(F_j, pid, \delta, sigma, once, retrigger)$ in $Current$ at the start of the loop .4–.12 to $\mathrm{dom}\, \sigma$ if $once = false$, or to $\emptyset$ if $once = true$; call $D_k^{j i_0}$ this domain, where j is the index of F_j, and i_0 is the beginning of σ. We claim that for all j and i_0, the subsequence of the $D_k^{j' i'_0}$ with $j = j'$ and $i_0 = i'_0$ is strictly increasing in $<_i$.

We show this by induction on computations. This is certainly true at the start of the loop .4–.12 when $i_0 = i$, since the corresponding subsequences with given j and i_0 are each of length one, and contain the sole domain $\emptyset$. (In particular, the claim holds when $i = 1$ at the start of this loop.) Then the effect of lines .4–.12 is to copy these domains from $Current$ to $Next$, keeping the order (because we use FIFO queues), possibly removing some domains (the killed entries), and possibly adding a finite number of copies of $D_k^{j i_0} \cup \{i\}$ at the immediate left of $D_k^{j i_0}$ (using advance at line .9). Note however that, because of Assumption 3.5, the latter operation must in fact add at most one copy of $D_k^{j i_0} \cup \{i\}$ to the immediate left of each $D_k^{j i_0}$. Note also that removing some domains preserves the strict ordering invariant. So we are left with proving that, given any sequence of domains $d_1 <_i d_2 <_i \ldots <_i d_N$, where $d_1, d_2, \ldots, d_N \subseteq [1, i-1]$, inserting $d_p \cup \{i\}$ immediately at the left of selected d_p's, $1 \leq p \leq N$, yields a $<_{i+1}$-strictly ordering sequence. It suffices to show that $d_1 \cup \{i\} <_{i+1} d_1 <_{i+1} d_2 \cup \{i\} <_{i+1} d_2 <_{i+1} \ldots <_{i+1} d_N \cup \{i\} <_{i+1} d_N$, i.e., that: (i) $d_p \cup \{i\} <_{i+1} d_p$ for every p, $1 \leq p \leq N$, and that: (ii) $d_p <_{i+1} d_{p+1}$ for every p, $1 \leq p < N$.

Let Z be any non-empty subset of $[i{+}1, +\infty)$, and K the cardinality of d_p. Then $d_p \cup \{i\} \cup Z$ and $d_p \cup Z$ agree on their K first elements (those of d_p), and their $(K + 1)$th elements are respectively i and the first element of Z, which is by assumption at least $i + 1$. So $d_p \cup \{i\} \cup Z <_{lex} d_p \cup Z$. Since Z is arbitrary, (i) holds.

Since every non-empty subset of $[i{+}1, +\infty)$ is also a non-empty subset of $[i, +\infty)$, it is clear that $d_p <_i d_{p+1}$ implies $d_p <_{i+1} d_{p+1}$, whence (ii) holds.

This finishes to show that at turn i of the loop .4–.13, for any fixed j and i_0, the $D_k^{j i_0}$'s are sorted in strictly increasing $<_i$ order in $Current$. So the first reported run is not only of minimum span $[i_0, i_\ell]$, but also $<_{i_{\ell+1}}$-minimal. But it is easy to see that if D and D' have the same span $[i_0, i_\ell]$ and $D <_{i_{\ell+1}} D'$, then $D <_{lex} D'$. So the first reported run is shortest. $\square$

The algorithm is still sound and complete on infinite logs, contrarily to Algorithm 2.5: we never had to assume S finite. The following is also clear, and says that runs are reported as soon as possible:

Theorem 3.12 (On-line) *The algorithm is Definition 3.9 is* on-line*: if there is a run of F_j against S with beginning i_0, and the span of its shortest run is $[i_0, i_\ell]$, then this is reported when reading record S_ℓ, i.e. when $i = i_\ell$ in the call to* `advance` *at line .9.*

3.3 Optimizations

There are two places where we can, and must, improve the efficiency of the algorithm of Definition 3.9.

First, at line .11, notice that g is false, but it might even be the case that we can statically determine (i.e., before we run the model-checker) that if g is false now, then it will remain false forever. Consider a typical *timeout* condition: one field `date` of the records S_i is a monotonically increasing time field, e.g., $i \leq i'$ implies S_i.`date`$\leq$ $S_{i'}$.`date` (such information can easily be supplied to the model-checker), and the transition δ is of the form $q \xrightarrow[g]{P} q'$, where P contains a row `date` $= \xi$, and g is the condition $\xi < t_0$, where t_0 is a time constant. The idea is that this transition should only be active while the current time has not reached t_0 yet, so attacks extending farther away in time should be discarded. As stated in Definition 3.9, the algorithm won't discard these transitions, because it is not aware that once g fails, it must fail forever—i.e., g is an *antitonic* function (provided Booleans are ordered by $false < true$), meaning that increasing the values of free variables can only make g decrease. Call a flexible variable ξ *increasing* in F if and only, whenever there is a row $id = \xi$ in F, id is a known monotonically increasing field. The precise optimization is that we may replace the marked $true^*$ test of line .11 by a test **not** `will_always_fail`(g), where `will_always_fail`(g) returns $true$ only when g is an antitonic function of the increasing flexible variables and of the counter variables (which can only increase through time), and is constant as a function of the non-increasing flexible variables. (Note that g may depend on rigid variables, which do not vary, at will.) For short, say that g is *antitonic*. Testing antitonicity is easily computed before the algorithm even starts, by using standard abstract interpretation techniques [3].

Second, look at line 7 of `advance`. This places thread pid in a position where it will wait for some record $S_{i'}$, $i' \geq i$ at which the transition $\delta \hat{=} q \xrightarrow[g]{P} q'$ can be fired (line .9). Now assume that δ may be fired either at $S_{i'}$ or at $S_{i''}$, with $i \leq i' < i''$. In general, it might be that, although δ can be fired at $S_{i'}$, this firing will never lead to a complete run, and we have to wait for the firing at $S_{i''}$ to obtain one. This is why we re-enqueue at line .10. However, this is not needed in special cases. For example, assume that $g = true$ and P binds no variable. (The variables *bound* by P are those at the right of the equals sign in rows.) Then any run that we get by firing $S_{i''}$ would be of the form $((q_1, i_1, \rho_1), \ldots, (q_{k-1}, i_{k-1}, \rho_{k-1}), (q_k, i_k, \rho_k), (q_{k+1}, i_{k+1}, \rho_{k+1}), \ldots)$ where $i_{k-1} \leq i < i_k = i'' < i_{k+1}$. Now firing $S_{i'}$ instead does not change the values of variables, since P binds no variable, hence $((q_1, i_1, \rho_1), \ldots, (q_{k-1}, i_{k-1}, \rho_{k-1}), (q_k, i', \rho_k), (q_{k+1}, i_{k+1}, \rho_{k+1}), \ldots)$ is another, strictly shorter run. So it was not necessary to re-enqueue at line .10. In this case, we set $retrigger$ to $false$ at line 7 of `advance`; i.e., we replace $true^*$ there by a call to some Boolean function `backtrack_needed`(P, g) that returns $false$ in this special case. This argument can be generalized to the case where P may bind any flexible variable, the rigid variables that P binds are either in $\mathrm{dom}\, \rho$ (i.e., they cannot change their values because of later calls to $Match$) or are not free in any guard labeling any transition reachable from q' (i.e., although they may change, they won't influence any later evaluation of a guard), and any reachable guard g' is antitonic (i.e, evaluating g at $S_{i'}$ instead of $S_{i''}$ can only make it truer). If so, `backtrack_needed`(P, g) returns $false$, otherwise $true$.

Apart from these two optimizations, we use a few other tricks. The set $Kill$ is regularly purged of pids of threads that have been killed already. Moreover, memoization is used extensively to avoid recomputing either the values of guards or the environments resulting from calls to $Match$—in fact, even resulting from individual row matching operations. Finally, there is no need to propagate whole partial runs σ in queue entries: changing the definition of ρ_f to map flexible variables ξ to *histories*, i.e., sequences of values of ξ along the run, and reserving a field name `line` holding line numbers i in records S_i allows one to just keep one environment from which the whole run can be reconstructed.

4 Conclusion

The algorithm of Definition 3.9 was implemented in the `logweaver` tool, together with the optimizations of Section 3.3, by the second author. Practical results look promising: for example, on a 800 MHz Pentium III (Coppermine) machine with 256Kb cache running Linux 2.2.14-5.0, simultaneously detecting probing attacks (the same source connects to several destination addresses, at least 100 times, all related connections being collected even beyond the first 100) and spoofing attacks (several sources connect to the same destination address) takes 0.34s. on a 942K firewall log consisting of

3026 lines, and 3.79s. on a 9731K firewall log consisting of 31437 lines. In the latter experiment, parsing the log itself already accounts for 2.05s., i.e., 54% of the time. Moreover, times are proportional to the size of the log, but only less than proportional to the size and number of signatures (because of memoization, which shares computations between signatures). Other small experiments conducted in Linux `syslog` and `messages` auditing report negligible auditing times. In most cases, the optimizations of Section 3.3 turned out to be crucial.

We do not claim that Algorithm 3.9 is the final solution to log auditing. There are cases where combinatorial explosion occurs; after all, this is an exponential-time algorithm. (Note that, although the complexity of the shortest runs problem is in NP, we conjecture that it is not NP-complete.) Nonetheless, in all cases until now, combinatorial explosions were due to errors in writing signatures. These errors can be located by inspection of execution traces of the model-checker, which our implementation produces on demand.

All in all, we join A. Mounji [13] in estimating that detecting complex correlations of events is feasible, even perhaps with efficiency comparable to much simpler intrusion detection systems that only do filtering and counting. Our import, as we hope to have demonstrated, is that it is even possible to detect complex correlations of events, on-line, using a very declarative style of signatures, expressed in a suitable variant of temporal logic. This temporal logic has a clean semantics and a clear notion of shortest run. This clarity fosters optimizations in the model-checker, ensures optimality with respect to minimizing spurious attack reporting (Theorem 3.8), and at the same time provides for better readability and maintainability of signature files.

Acknowledgments

We thank Xiaobo Li, who improved and maintained the implementation of the model-checker of Section 2, and Stéphane Demri for his suggestion that model-checking the logic of Section 2 was NP-complete.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[2] T. Christiansen, L. Wall, and J. Orwant. *Programming Perl*. O'Reilly and Associates, 3rd edition, 2000.

[3] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. Correct version at `http://www.dmi.ens.fr/~cousot/COUSOTpapers/JLP92.shtml`.

[4] M. Crosbie, B. Dole, T. Ellis, I. Krsul, and E. Spafford. IDIOT – users guide. Technical report, Purdue University, COAST Laboratory, 1996.

[5] M. Ducassé and J.-P. Pouzol. Handling generic intrusion signatures is not trivial. In *Recent Advances in Intrusion Detection (RAID) Workshop*, 2000.

[6] S. T. Eckmann, V. Giovanni, and R. A. Kemmerer. STATL: An attack language for state-based intrusion detection. In *ACM Workshop on Intrusion Detection Systems*, 2000.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[8] S. Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Department of Computer Science, Purdue University, West Lafayette, 1995.

[9] U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, 1999.

[10] LSV. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer Verlag, 2001.

[11] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1991.

[12] M. Minoux. LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Information Processing Letters*, 29(1):1–12, 1988.

[13] A. Mounji. *Languages And Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, FUNDP, Namur, Belgium, 1997.

[14] V. Paxon. BRO: A system for detecting network intruders in real-time. In *7th USENIX Security Symposium*, 1998.

[15] P. A. Porras. STAT – a state transition analysis tool for intrusion detection. Master's thesis, University of California, Santa Barbara, 1992.

[16] M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a generalized tool for network monitoring. In *11th Systems Administration Conference (LISA'97)*, pages 1–8. USENIX Association, 1997.

[17] M. Roesch. Snort: Lightweight intrusion detection for networks. In *13th Systems Administration Conference (LISA'99)*, pages 229–238. USENIX Associations, 1999.

[18] E. Shapiro. Alternation and the computational complexity of logic programs. *Journal of Logic Programming*, 1(1):19–33, 1984.

[19] A. Sistla and E. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[20] H. Spencer. regexp package. Available at `http://arglist.com/regex/`, 1986.

[21] M. Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, pages 238–266. Springer-Verlag LNCS 1043, 1996.

[22] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.