Future Generation Computer Systems 113 (2020) 196–206

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Modeling and efficiently detecting security-critical sequences of actions☆

Antonella Guzzo *,[1], Michele Ianni[1], Andrea Pugliese[1], Domenico Saccà[1]

University of Calabria, Italy

ARTICLE INFO

Article history:
Received 25 November 2019
Received in revised form 4 May 2020
Accepted 26 June 2020
Available online 6 July 2020

ABSTRACT

Many different techniques and tools have been proposed to prevent and detect malicious activities by means of a very challenging data analysis task. The main sources for data analysis are the activity logs that are produced in large volumes at run time and are often characterized by semantically rich properties. The paper proposes a framework for analyzing the collected logs in order to provide the defenders with relevant insights on the attacks that have been conducted. The framework consists of two ingredients: (*i*) a modeling language to define the patterns of attacks that are of interest to the defenders, and (*ii*) an algorithm that is able to identify, in an input log, all possible attacks conforming to the given patterns. The paper presents a formalization of the modeling language and a study of its properties from a theoretical viewpoint as well as the algorithm (along with an ad-hoc data structure) that is proven to be very efficient in the identification of attacks in real world scenarios.

1. Introduction

An impressive number of security vulnerabilities have been discovered in the last few years that can be exploited by security attacks and, therefore, different techniques have been proposed to prevent and detect malicious activities. Indeed, designing mechanisms and tools to protect information systems is an active research topic, with relevant social and economic impacts and a multidisciplinary approach [1–4]. As a matter of fact, such tools are increasingly adopted in enterprises and they are collecting a large volume of data that must be analyzed to discover additional information and useful insights on the behavior of malevolent users.

The task of detecting malicious activities is very challenging. The logs that have to be analyzed are of a semantically rich nature, as they mix actions performed by attackers with actions performed by normal users. This calls for defining mechanisms that are able to automatically filter the various sequences of actions and focus on the critical ones. Moreover, it must be observed that, together with their malevolent activities, attackers usually carry out some "legitimate" activities. This means that a pattern of attack is typically "hidden" within a longer sequence of actions, from which an attack may become hard to be detected.

The problem related to the identification of malicious activities has been already tackled in the literature from various perspectives. This lead to the birth of different kinds of systems, ranging from *honeypots* to *Intrusion Detection Systems*. Abstracting from their technical peculiarities, such systems are designed to analyze specific data sources and to deal with some specific malicious behaviors. In fact, enriching the flexibility and the expressiveness of such systems is an important research issue. In this paper, we face this issue and we propose an approach that is able to support the security domain expert in the task of identifying *any kind* of (malicious) behavior s/he is interested in from logs or real-time streams of recorded actions.

Our approach consists of two salient ingredients. The first ingredient is the use of a simple, yet expressive modeling language to formalize the attack patterns that are of interest. The features of the modeling language have been carefully designed to allow defenders to specify the activities of interest while retaining nice computational properties on top of which efficient static optimization methods can be implemented (for instance, to check the consistency of the specification or for their minimization/optimization). We now present an example specification that will be used as a running example throughout the paper.

Example 1. Consider the patterns of attack depicted in Fig. 1, according to an intuitive notation that we are now going to illustrate.

Basically, attacks are represented as a graphical model, which we name *sequence model*. The model in the figure represents the

☆ Part of this work has been funded by the Italian Ministry of Economic Development grants "EMPHAsis" and "ProtectID".

* Corresponding author.
E-mail addresses: antonella.guzzo@unical.it (A. Guzzo), michele.ianni@unical.it (M. Ianni), andrea.pugliese@unical.it (A. Pugliese), domenico.sacca@unical.it (D. Saccà).
[1] All authors have contributed equally to the manuscript.

https://doi.org/10.1016/j.future.2020.06.054
0167-739X/

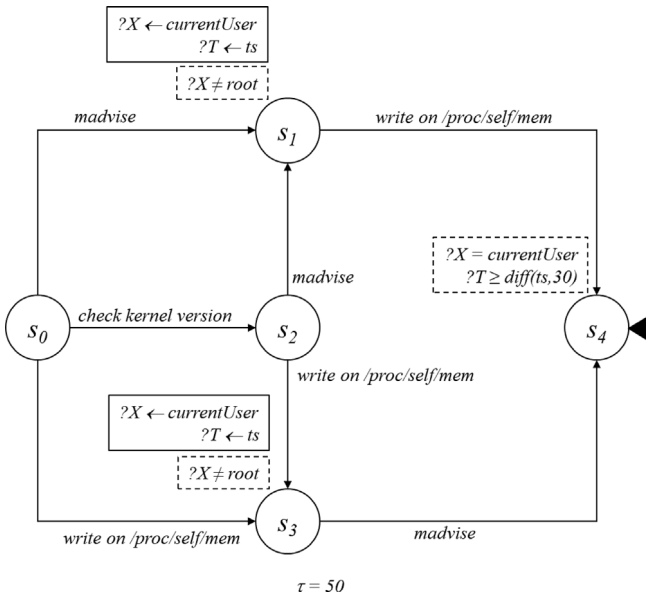


Fig. 1. The sequence model M used as running example.

exploitation of a popular security vulnerability of the Linux kernel named *Dirty COW* (Dirty Copy-On-Write, CVE-2016-5195). Using the vulnerability, an attacker tries to perform a local privilege escalation by exploiting a race condition in the implementation of the copy-on-write mechanism in the kernel's memory management subsystem. In particular, the attack works by concurrently performing several writes on "/proc/self/mem" and several "madvise" calls, in any order, possibly after having checked the version of the kernel on the machine. In the model, this sequence of actions is captured by paths through the states of the model. In particular, the sequence of actions that are performed along the path s_0, s_1, s_4 represents the *target sequence* where the call to "madvise" is performed first; the path s_0, s_3, s_4 identifies instead the target sequence where the "write on /proc/self/mem" is performed first, and the other paths that go through s_2 represent the cases where the attacker first checks the kernel version.[2] State s_4 is the one where an instance of the model ends (this is represented by the black arrow in the picture).

The annotations on the states describe the use of variables to check additional constraints on the sequences of actions of interest for the defender: (i) variable $?X$ is used to keep track of the identity of the user performing the actions by setting its value in states s_1 or s_3 after having checked that the user is not "root" and by later requiring, on state s_4, the actions to be performed by the same user; (ii) variable $?T$ is used to ensure that each pair of actions is performed within 30 time units, by first setting its value in states s_1 or s_3 and, then, checking the value in state s_4. An overall time constraint, which limits the maximum duration of an instance of the model, is expressed by threshold τ — in the example, the maximum allowed duration is 50 time units.[3] □

As emerged from the above example, *sequence models* formalize patterns of attack by allowing defenders to succinctly /implicitly describe sequences of actions (which we call *target sequences*) that are crucial to detect. In fact, the goal of the analyst (and, ultimately, of the defender) is to identify such target sequences from a given log, even when they are interleaved with the execution of legitimate actions that are not related to the attack being conducted.

Example 2. We first recall from Example 1 that s_0, s_3, s_4 identifies the target sequence where the attacker first performs "write on /proc/self/mem" and then the call to "madvise". Then we point out that a log hardly happens to precisely consist of a sequence with these two actions only for other actions may precede or follow or interleave them.

Observe that, although actions different from "write on /proc/self/mem" and "madvise" may be legitimate, we cannot just skip or ignore them while analyzing a given log as they can instead play a crucial role for detecting other malicious sequences. □

The second ingredient of our framework is an algorithm that efficiently carries out the activity detection task we have sketched in the above example; that is, given as input a log L produced during a working session and a sequence model M specified by the defender, the algorithm is capable to identify *all* target sequences that are actually hidden in L. To this end, the algorithm uses an ad-hoc data structure, called *sequence tracking index*, along with a method for maintaining the index after the insertion of a new tuple in the log. Because of this peculiarity, the algorithm can be used to detect attacks not only when logs have been entirely stored, but also *on-the-fly*, i.e., *while* the attacker is interacting with the system.

Organization. The rest of the paper is organized as follows. Works that are related to our paper are illustrated in Section 2. The notions of sequence models and target sequences are formalized in Section 3. The theoretical analysis of the properties of our formal language is reported in Section 4. Section 5 presents our algorithmic approach to identify target sequences, together with a discussion of our experimental validation. Finally, conclusion and few avenues for further research are discussed in Section 6.

2. Related work

Several approaches to the problems of modeling and detecting malicious behavior have been proposed so far. This section will only discuss the relevant literature that is the closest in spirit to our approach.

A large number of monitoring systems for automatic large scale attack detection have been proposed, among which intrusion detection techniques based approaches [5,6] and honeypot based approaches [7–9] are continuously evolving with broad potential. Irrespective of the specific strategy or technique adopted, we are interested in the above systems as sources of huge amounts of data (comprising malicious activities and information about the attackers and their patterns of actions) that, even with their heterogeneous structure and semantic nature, can be profitably used in our approach as input for the detection task. In fact, our detection technique is not targeted to specific data formats and can be useful as a complement of the existing detection systems.

The idea of comparing a model behavior (the "normative" model) against the observed behavior registered in a log has been widely explored in the area of Business Process Management [10]. There, a number of works have been proposed focusing on the so-called *conformance checking* task [11,12], which relates and compares events recorded during the executions of a process to the actions belonging to a process specification. At a conceptual level, our approach is aimed at identifying the logs that

[2] It should be observed that, in order to capture the full-fledged exploitation of the vulnerability, a number of instances of the model have to be chained together — for simplicity, we do not discuss this in the running example.

[3] A slightly extended version of the model, which represents the user moving to "root", is reported in Appendix.

fit the sequence model and, therefore, it is specular to conformance checking which is instead aimed at detecting the logs that *do not* fit the model (and also to quantify the "degree" of difference between logs and models, according to some formal metric). Nonetheless, we believe that our results can be used for conformance checking — this is a topic that constitutes an interesting avenue for further research. In fact, it must be noticed that conformance checking techniques have been developed for different modeling languages (used to specify the underlying process), ranging from classical graph-based representations such as Heuristic Nets and AND-OR graphs, to semantically richer models such as Petri nets, causal nets, or logic-based models [10,13,14]. Differently from our approach, such earlier works hardly considered attributes and constraints, except for the classical ones representing the ordering between actions. Our approach also distinguishes itself for considering a rather articulated abstraction for action sequences – which nicely fits a number of application domains – whereas classical conformance checking techniques deal with "traces" which are just seen as sequences of action names, possibly equipped with timestamps.

The use of graph-based models to describe (with various efficiency/expressiveness tradeoffs) the structure of security activities of interest is also well established [15–20]. The *attack graphs* that constitute the basis of some approaches [15,16] are generally constructed by analyzing the dependencies among vulnerabilities and security conditions that have been identified in a target network. Recent works on the detection instances of graph-based models in sequences of logged actions have shown that acceptable detection times in real-world cases can be obtained by using index structures that limit the number of partial instances to be maintained. These structures usually exploit the various kinds of constraints available to only look at the set of (partial) instances that lie within a fixed temporal window [16,17] An indexing technique for alert correlation is proposed in [17] that supports DFA-like patterns with user-defined correlation functions. Hypergraph-based models are proposed in [19], whereas [20] looks for "unexplained" graph-modeled activities, i.e., activities that do not fit any of the given models.

Another related research area is that of *Complex Event Processing* (CEP) systems. Such systems usually aim at extracting patterns of streamed events that match incoming events on the basis of their content and on some ordering relationships [21]. CEP-based infrastructures have been used in a large number of domains, including financial trading, IoT monitoring, and social/sensor network analysis [21,22]. A typical CEP infrastructure in state-of-the-art systems uses a distributed engine organized as a set of event brokers (or event processing agents), connected within an overlay network (the event processing network) and implementing specialized routing and forwarding functions, with scalability as their main concern. In fact, scalability and other practical issues like fault tolerance and distribution are the primary focus in the design of CEP systems, with the objective of making them usable in real-life scenarios [21]. At the front-end level, CEP systems adopt different query languages without simple formal denotational semantics [23,24].

Similarly to CEP systems, the primary goal of our proposed techniques is to detect the occurrence of "complex events" in a stream/sequence of data. However, our solution differs significantly, both conceptually and from the technical viewpoint.

First of all, in a CEP system, the user is asked to explicitly describe each pattern she is interested in, for instance, in terms of a rule-based specification language. In our approach, patterns of interests are specified by means of a rich graphical notation, the sequence model, in which each possible path can be viewed as a distinguished query for a CEP system. On the one hand, the graph modeling appears better suited to describe the possible attacks by a malicious user — sequence models directly describe the behaviors of attackers and, hence, they better adhere to specific application domains from the knowledge representation viewpoint. On the other hand, while any sequence model can be transformed into a set of patterns by explicitly listing all possible paths occurring in it, it must be observed that this process would immediately result in an exponential blowup of the specification (since directed graphs can succinctly encode exponentially many directed paths). In other words, our query language based on sequence models is exponentially more succinct than classical query-based languages for CEP systems (even when they are equipped with rich syntactic features allowing to define complex patterns in terms of disjunctions, conjunctions and negations of simpler patterns). This succinctness obviously called for more specialized algorithms and approaches that found no counterpart in the CEP literature.

The second important distinguishing features of our approach compared to CEP systems is related to the semantics of the "query" specification language. In our approach, indeed, we are interested in finding all possible occurrences of some given sequence in the log at hand — this is clearly motivated by the specific application domain, where we must be very cautious when analyzing and inspecting suspicious users' operations. As a result, our techniques must adopt rather complex data structures that are able to maintain, in a dynamic fashion, all possible active paths of attack till we can safely disregard them or advert that some malicious activity is occurring. Differently from this very cautious approach, CEP systems typically adopt different semantics for matching the query against a data stream. For instances, in some cases they just look for patterns occurring over contiguous time points. More flexible systems allow – as in our approach – interleaving relevant events with irrelevant ones, but in these cases they are designed to identify whether one occurrence can be found rather than identifying all possible occurrences. In particular, the occurrence returned to the user is selected according to some greedy heuristics designed to speedup the computation. This is clearly not an ideal feature when analyzing possible attacks, as the security expert might (non-deterministically) miss important insights on users' behaviors.

Finally, a further – while minor – distinguishing feature of our approach is that it is designed to reason about timestamps as "first-class citizens", as they are indeed a part of the specification language itself. In CEP systems, instead, query languages are typically defined to reason about streaming events data only, with timing information being delegated to the specific semantics being adopted for the evaluation.

3. Sequence models

We assume the existence of the following mutually disjoint sets: (i) a set $\texttt{Act}$ of actions symbols, (ii) a set $\texttt{Var}$ of variable symbols, (iii) a totally-ordered domain $\texttt{T} \subseteq \mathbb{N}$ of timestamps, and (iv) a set $\texttt{Att} = \{A_1, \ldots, A_h\}$ of attributes; in particular, for each $j \in \{1, \ldots, h\}$, the attribute A_j is associated with a domain of values and a name, respectively denoted as $dom(A_j)$ and $name(A_j)$.

Definition 1 (*Log*)**.** A *log* is a sequence $L = \ell_1, \ldots, \ell_n$, with $n > 0$, where each ℓ_i is a tuple $\langle \ell_i.act, \ell_i.ts, \ell_i.name(A_1), \ldots, \ell_i.name(A_h)\rangle$ such that:

- $\ell_i.act \in \texttt{Act}$ is the action registered by the tuple;
- $\ell_i.ts \in \texttt{T}$ is the timestamp at which the action occurs;
- for each $j \in \{1, \ldots, h\}$, $\ell_i.name(A_j) \in dom(A_j)$ is an attribute value taken from $dom(A_j)$ and encoding some feature of interest.[4]

[4] We sometimes use "virtual" attributes, defined as functions over "real" ones — since the two kinds of attributes are completely equivalent in the framework, we treat them in the same way in order to keep the notation uniform.

Hereinafter we assume, without loss of generality, that $\ell_1.ts = 0$ holds; moreover, for each $i, k \in \{1, \ldots, n\}$ with $i < k$, it is assumed that $\ell_i.ts < \ell_k.ts$, which means that actions are not executed simultaneously.[5]

Let $?X \in \texttt{Var}$ be a variable. Then, an expression having the form $[?X \leftarrow name(A_j)]$ is called a *(variable) assignment*, whereas an expression having the form $[?X\ op\ u]$, where $op \in \{=, \neq, <, >, \leq, \geq\}$ and $u \in \{name(A_1), \ldots, name(A_h)\} \cup \bigcup_j dom(A_j)$, is called a *constraint*.

Definition 2 (*Sequence Model*)**.** A *sequence model* is a tuple $M = \langle S, s_0, F, \delta, \texttt{assign}, \texttt{constr}, \tau \rangle$ where:

1. S is a set of *states*;
2. $s_0 \in S$ is the *initial state*;
3. $F \subseteq S$ is the set of *final states*;
4. $\delta \subseteq (S \setminus F) \times \texttt{Act} \times (S \setminus \{s_0\})$ is the set of *stage transitions*; let $g(M) = (S, \delta)$ denote the directed graph over the nodes in S and whose edges correspond to the stage transitions in δ — it is required that $g(M)$ is acyclic;
5. $\texttt{assign}$ is a function that associates a set of assignments with every state in $S \setminus \{s_0\}$;
6. $\texttt{constr}$ is a function that associates a set of constraints with every state in $S \setminus \{s_0\}$;
7. $\tau \in \mathbb{N}$ is the *maximum duration*.

Example 3. In the sequence model M of Fig. 1, variable assignments are depicted in solid boxes, constraints are depicted in dashed boxes, and a black arrow annotates the final state. Moreover:

- $S = \{s_0, s_1, s_2, s_3, s_4\}$;
- $F = \{s_4\}$;
- $\delta = \{(s_0, madv, s_1), (s_0, ckv, s_2), (s_0, wop, s_3), (s_2, madv, s_1), (s_2, wop, s_3), (s_1, wop, s_4), (s_3, madv, s_4)\}$[6];
- $\texttt{assign}(s_1) = \texttt{assign}(s_3) = \{?X \leftarrow currentUser, ?T \leftarrow ts\}$;
- $\texttt{constr}(s_1) = \texttt{constr}(s_3) = \{?X \neq root\}$; $\texttt{constr}(s_4) = \{?X = currentUser, ?T \geq diff(ts, 30)\}$;
- $\tau = 50$. □

A *substitution* θ is a function mapping every variable $?X \in \texttt{Var}$ to some constant $\theta(?X) \in \bigcup_j dom(A_j)$. A *path* $s_0, s_1, \ldots, s_m$ in $g(M)$, which originates from the initial state s_0, is *complete* if $s_m \in F$.

Definition 3 (*Supported Log and Target Sequence*)**.** A log $L = \ell_1, \ldots, \ell_n$ is *supported* by a sequence model $M = \langle S, s_0, F, \delta, \texttt{assign}, \texttt{constr}, \tau \rangle$ if there is a substitution θ, a complete path $s_0, s_1, \ldots, s_m$, and m indices $1 \leq k_1 < \cdots < k_m \leq n$ such that:

1. $(s_{i-1}, \ell_{k_i}.act, s_i) \in \delta$, for each $i \in \{1, \ldots, m\}$;
2. $?X \in \texttt{Var}$ and $\theta(?X) = \ell_{k_i}.name(A_j)$ holds, for each $i \in \{1, \ldots, m\}$ and $[?X \leftarrow name(A_j)] \in \texttt{assign}(s_i)$;
3. if u is an attribute value, then $?X \in \texttt{Var}$ and $\theta(?X)\ op\ u$ holds, for each $i \in \{1, \ldots, m\}$ and $[?X\ op\ u] \in \texttt{constr}(s_i)$;
4. if $u = name(A_j)$, then $?X \in \texttt{Var}$ and $\theta(?X)\ op\ \ell_{k_i}.name(A_j)$ holds, for each $i \in \{1, \ldots, m\}$ and $[?X\ op\ u] \in \texttt{constr}(s_i)$;
5. $\ell_{k_m}.ts - \ell_{k_1}.ts \leq \tau$.

If the above conditions hold, then we also say that $\ell_{j_1}, \ldots, \ell_{j_m}$ is a *target sequence* of M in L.

[5] Indeed, these assumptions might be enforced by taking a sufficiently fine-grained time domain, eventually scaled to the domain of the natural numbers.

[6] Here, "madv" stands for "madvise", "ckv" stands for "check kernel version", and "wop" stands for "write on /proc/self/mem" — we will sometimes use these shorthand notations in the rest of the paper.

Table 1
Example log L.

	act	*ts*	*currentUser*
ℓ_1	*madvise*	0	*user*1
ℓ_2	*check kernel version*	5	*root*
ℓ_3	*madvise*	10	*root*
ℓ_4	*check kernel version*	15	*user*2
ℓ_5	*write on /proc/self/mem*	20	*root*
ℓ_6	*write on /proc/self/mem*	25	*user*2
ℓ_7	*other action*	30	*user*2
ℓ_8	*check kernel version*	35	*user*2
ℓ_9	*madvise*	40	*user*2
ℓ_{10}	*write on /proc/self/mem*	45	*user*2
ℓ_{11}	*write on /proc/self/mem*	50	*user*1

The set of all logs supported by M is denoted by $\mathcal{L}(M)$.

Example 4. Consider the sequence model M of Example 1 and the log L depicted in Table 1.

L is supported by M and the full set of target sequences of M in L consists in the following sequences:

- ℓ_4, ℓ_6, ℓ_9;
- ℓ_6, ℓ_9;
- $\ell_4, \ell_9, \ell_{10}$;
- ℓ_9, ℓ_{10}. □

4. Theoretical characterization

In this section, we study and characterize three problems that arise with respect to our proposed notions of sequence model and target sequence: checking the consistency of a given sequence model, checking its minimality, and finding all target sequences of the model in a given log.

Throughout the section, we assume that $M = \langle S, s_0, F, \delta, \texttt{assign}, \texttt{constr}, \tau \rangle$ is a given sequence model with its associated graph $g(M)$ (see Definition 2).

4.1. Consistency of sequence models

In order to study the consistency of sequence models, we start by introducing the notion of well-formed models.

Definition 4 (*Well-Formed Sequence Model*)**.** We say that M is *well-formed* if the following conditions hold:

(a) If s and s' are two states in S, $[?X \leftarrow name(A_j)] \in \texttt{assign}(s)$, and $[?X \leftarrow name(A_{j'})] \in \texttt{assign}(s')$, then s and s' are not reachable from each other in the graph $g(M)$; that is, after variable $?X$ takes some value in a state, then no subsequent state can overwrite that value.

(b) For each $s \in S$, if $[?X\ op\ u] \in \texttt{constr}(s)$, then each path in $g(M)$ connecting s_0 to s includes a node s' such that $?[X \leftarrow name(A_j)] \in \texttt{assign}(s')$; that is, before being used in some constraint, variables must be initialized.

(c) There is at least one path $s_0, s_1, \ldots, s_h$ in $g(M)$ such that $s_h \in F$ and $h - 1 \leq \tau$; that is, there is at least one path that can be executed within the maximum duration τ.

(d) If s and s' are two states in S such that there is a path in $g(M)$ from s to s', $[?X\ op\ u] \in \texttt{constr}(s)$, and $[?X\ op'\ u'] \in \texttt{constr}(s')$, then $u'op\ u$ and $u\ op'u'$ must hold; that is, constraints never contradict each other.

Theorem 1. *Deciding whether M is well-formed is feasible in time $O(|S|^3)$.*

Proof. In order to check (a), we perform a depth-first visit of the graph $g(M)$ starting at the initial state s_0. We have to check that after some state has assigned a value to some variable $?X$, then no subsequent state in the visit exists overwriting that value. The algorithm overall takes $O(|S|^2)$ steps. In order to check (b), we first build the graph G' by reverting the edge orientation of $g(M)$. Then, for each s such that $\texttt{constr}(s) \neq \emptyset$, we visit (again, depth-first) the graph G' in order to check that the variables involved in the constraints are initialized at some point. This takes $O(|S| \times |S|^2)$. Condition (c) reduces to computing a minimum-weighted path from any of the initial state to any of the final states, which is feasible in $O(|S|^2)$. Eventually, if $s_0, s_1, \ldots, s_h$ is the path that is computed, then we just have to check that $h - 1 \leq \tau$ holds. Finally, condition (d) needs to iterate over all possible pairs of states, hence taking $O(|S|^2)$. □

The consistency problem consists in deciding if a sequence model M has at least one target sequence. The following result gives us a sufficient condition.

Theorem 2. *If M is well-formed, then $\mathcal{L}(M) \neq \emptyset$.*

Proof. Consider a path $s_0, s_1, \ldots, s_h$ in $g(M)$ where $s_h \in F$ and $h - 1 \leq \tau$, which exists because of condition (c). For each $i \in \{1, \ldots, h\}$, let a_i be the action such that $(s_{i-1}, a_i, s_i) \in \delta$. Then, let us define a log $L = \ell_1, \ldots, \ell_h$ and a substitution θ as follows. First, we set $\ell_i.act = a_i$ and $\ell_i.ts = i - 1$, for each $i \in \{1, \ldots, h\}$. Then, let V be the set of variables $?X$ such that $[?X \leftarrow name(x)]$ occurs in $\texttt{assign}(s_i)$, for some state $s_i \in \{s_1, \ldots, s_h\}$. For each variable $?X \in V$, let c_x be some value satisfying all constraints $[?X \ op \ y] \in \texttt{constr}(s_i)$ defined on that variable on the given path. Because of condition (d), note that c_x is well-defined. Then, we define θ as the function mapping such variables $?X \in V$ to c_x. By exploiting (a) and (b), it can be easily checked that θ witnesses that L is supported by M. □

It should be observed that meaningful models are *only* those with $\mathcal{L}(M) \neq \emptyset$ — otherwise, the use of M to evaluate target sequences is unproductive as every sequence is trivially considered non-target. We believe that Theorems 1 and 2 are encouraging results for further research, as they outline ways to exploit features of the model for developing efficient strategies to tackle the consistency problem.

4.2. Structural minimality of sequence models

Addressing the structural minimality problem amounts to finding out redundancies in sequence models that can be dropped out as they do not actually provide any additional information about the permitted and forbidden sequences.

In the following, we say that M is *connected* if it is well-formed and every state in it occurs in some complete path.

Observation 1. *Assume that M is well-formed and let M' be the model derived from M by removing all states $s \in S$ that do not occur in any complete path. Then, M' is connected and $\mathcal{L}(M) = \mathcal{L}(M')$.*

A *retract* on M is a function $f : S \mapsto S$ such that:

- $f(s_0) = s_0$;
- for each state $s \in S$:
  - $f(s) = f(f(s))$;
  - $f(s) \in F$ iff $s \in F$;
  - $\texttt{assign}(f(s)) = \texttt{assign}(s)$, $\texttt{constr}(f(s)) = \texttt{constr}(s)$;
  - $(f(s), a, f(s')) \in \delta$ iff $(s, a, s') \in \delta$, for each state $s' \in S$.

By slight abuse of notation, let $f(S) = \{f(s) \mid s \in S\}$. Moreover, let us define $M_{\downarrow f}$ as the sequence model $\langle S', s_0, F', \delta', \texttt{assign}', \texttt{constr}', \tau\rangle$ such that $S' = f(S)$, $F' = F \cap S'$, and δ', $\texttt{assign}'$ and $\texttt{constr}'$ are the restrictions over S' of the respective elements in M.

We say that M is *minimal* if it is connected and there is no retract f such that $f(S) \subset S$ and $\mathcal{L}(M) = \mathcal{L}(M_{\downarrow f})$.

Theorem 3. *Assume that M is connected. Let f be a retract such that $f(S) \subset S$ and $\mathcal{L}(M) = \mathcal{L}(M_{\downarrow f})$. Then, there is a retract f' on M such that $|f'(S)| = |S| - 1$ and $\mathcal{L}(M) = \mathcal{L}(M_{\downarrow f'})$.*

Proof. Given the retract f, let us take an arbitrary element $\bar{s} \in f(S) \setminus S$ and consider the function f' such that $f'(s) = s$ for each $s \neq \bar{s}$, and $f'(\bar{s}) = f(\bar{s})$. Note that $|f'(S)| = |S| - 1$. Moreover, it can be checked that f' is a retract and that, in fact, $\mathcal{L}(M) \supseteq \mathcal{L}(M_{\downarrow f'}) \supseteq \mathcal{L}(M_{\downarrow f})$. Since $\mathcal{L}(M) = \mathcal{L}(M_{\downarrow f})$ holds by hypothesis, we conclude that $\mathcal{L}(M_{\downarrow f'}) = \mathcal{L}(M)$. □

A retract f such that $|f(S)| = |S| - 1$ is said to be *simple*. In the light of the above result, in order to compute a minimal model we can just focus on simple retracts.

Theorem 4. *Assume that M is well-formed. Then, a minimal model M^* such that $\mathcal{L}(M) = \mathcal{L}(M^*)$ can be computed in polynomial time.*

Proof. We first modify M by removing some of its states in order to make it connected (cf. Observation 1). Then, we look for a pair of states s and s' in M, with $s \neq s'$, such that:

- $s' \in F$ iff $s \in F$;
- $\texttt{assign}(s') = \texttt{assign}(s)$, $\texttt{constr}(s') = \texttt{constr}(s)$;
- $(s', a, x) \in \delta$ iff $(s, a, x) \in \delta$;
- $(x, a, s') \in \delta$ iff $(x, a, s) \in \delta$.

The state s' is removed from M and the process is repeated till a fixpoint is reached. The resulting model M^* is minimal and $\mathcal{L}(M) = \mathcal{L}(M^*)$. Correctness follows by Theorem 3. □

4.3. Complexity of the identification problem

We now characterize the complexity of identifying all the target sequences of a sequence model in a log.

Theorem 5. *Given a log $L = \ell_1, \ldots, \ell_n$ and a sequence model $M = \langle S, s_0, F, \delta, \texttt{assign}, \texttt{constr}, \tau\rangle$, the problem of finding all the target sequences of M in L requires exponential time w.r.t. the cardinality of S.*

Proof (*Sketch*)**.** Given two positive integers $k > 1$ and $h > 1$, consider a log $L = \ell_1, \ldots, \ell_n$ with $n = k \cdot h$ and a sequence model $M = \langle S, s_0, F, \delta, \texttt{assign}, \texttt{constr}, \tau\rangle$ with:

- $S = \{s_0, s_1, \ldots, s_k\}$;
- $F = \{s_k\}$;
- $\delta = \{(s_{i-1}, act_i, s_i) \mid i \in [1, k]\}$;
- $act_{i-1} = act_i$ for all $i \in [1, k]$;
- $\texttt{assign}$ and $\texttt{constr}$ having empty domains;
- $\tau = \ell_n.ts$.

In this case, there exists a target sequence $\ell_{j_1}, \ldots, \ell_{j_k}$ for each possible set of indices $\{j_1, \ldots, j_k\}$ such that $1 \leq j_1 \leq h$, $h + 1 \leq j_2 \leq 2 \cdot h$, and so on (in general, $(i-1) \cdot h + 1 \leq j_i \leq i \cdot h$). Each j_i can therefore be drawn from a set of h integers — as a consequence, the number of target sequences of M in L is $h^k = h^{|S|-1}$. □

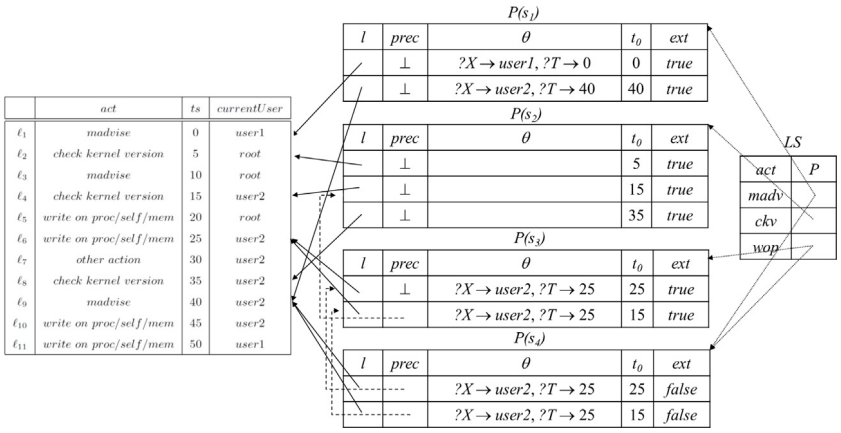


Fig. 2. ST-Index associated with the sequence model and log of our running example (up to tuple ℓ_9).

5. Efficient identification of target sequences

Theorem 5 states that the identification problem is intractable. In this section, we introduce a specifically-designed index structure along with an algorithm for maintaining the index after the insertion of a new tuple in the log, whose main objective is that of efficiently keeping track of partial target sequences in order to speed up the identification process. Then, we show the results of the experimental assessment we performed in order to appreciate the efficiency of the proposed index.

5.1. The sequence tracking index

Definition 5 (*Sequence Tracking Index (ST-Index)*)**.** Given a sequence model $M = \langle S, s_0, F, \delta, \texttt{assign}, \texttt{constr}, \tau \rangle$ and a log L, their associated *ST-Index* $\mathcal{I}_{M,L}$ is a tuple $\langle \mathcal{P}, LS \rangle$ where:

- $\mathcal{P}$ is a set containing a *state tracking table* $P(s)$ for each $s \in S$. Each table $P(s)$ contains rows of the form $\langle l, prec, \theta, t_0, ext \rangle$ where l is a pointer to a tuple in L, $prec$ is a pointer to a row in a state tracking table, θ is a variable substitution, t_0 is a timestamp, and $ext \in \{true, false\}$.
- LS is a *label-state table* containing rows of the form $\langle act, P \rangle$, where $act \in \texttt{Act}$ is an action symbol appearing in M and P is the set of pointers to all the tables $P(s) \in \mathcal{P}$ such that $(\cdot, act, s) \in \delta$.

Intuitively, in order to correctly index the target sequences of a model M in a log L, the ST-Index builds a set $\mathcal{P}$ of tables (one for each state in the model) whose rows contain the minimal information needed in order to track (possibly partial) target sequences. In particular, each row contains (i) a pointer to the log tuple of interest, (ii) a pointer to the table row that represents the partial target sequence to which the log tuple of interest can be added, (iii) the current variable substitution, (iv) the time point at which the partial sequence started, and (v) a flag indicating whether the partial sequence can be further extended. Table LS is used in order to quickly find the tables in $\mathcal{P}$ whose content can be affected by the insertion of a log tuple, based on the action reported in the tuple itself.

Example 5. Consider the sequence model and log of our running example. The contents of the ST-Index after indexing the log up to tuple ℓ_9 are depicted in Fig. 2. It should be observed that, by following backward pointers from the two rows in $P(s_4)$ we can immediately identify the target sequences ℓ_4, ℓ_6, ℓ_9 and ℓ_6, ℓ_9. □

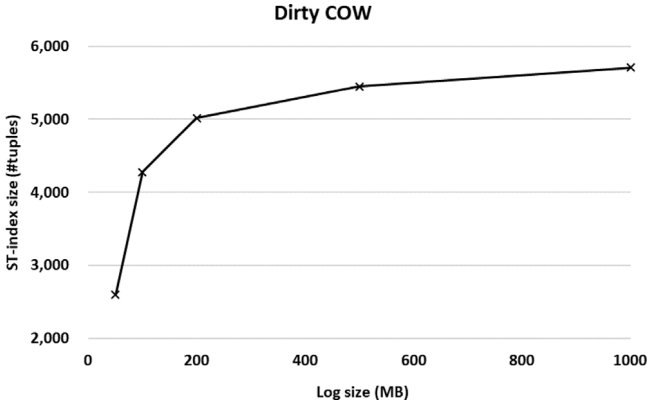


Fig. 3. Size of the ST-Index when varying log size (Dirty COW).

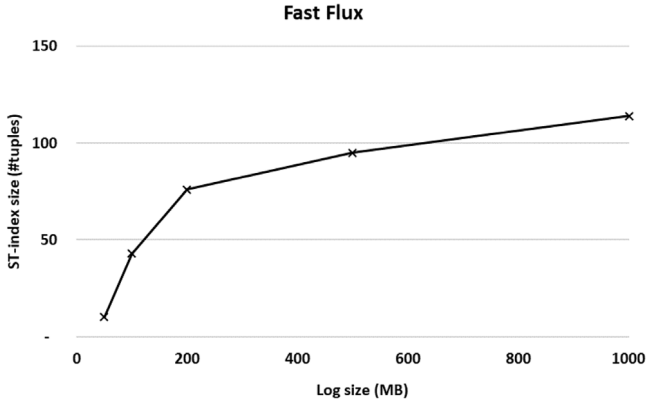


Fig. 4. Size of the ST-Index when varying log size (Fast Flux).

5.2. Insertion algorithm

Algorithm 1 is executed when a new log tuple must be added to an ST-Index. In the algorithm, $\bar{x}$ is a pointer to x.

Lines 6–11 check whether the new log tuple ℓ can be the first one of a target sequence — after applying the variable assignments, it checks whether the substitution satisfies the needed conditions and possibly adds a corresponding row to the state tracking table. Lines 13–22 check whether ℓ can extend an existing partial sequence — every time a table row is found that could

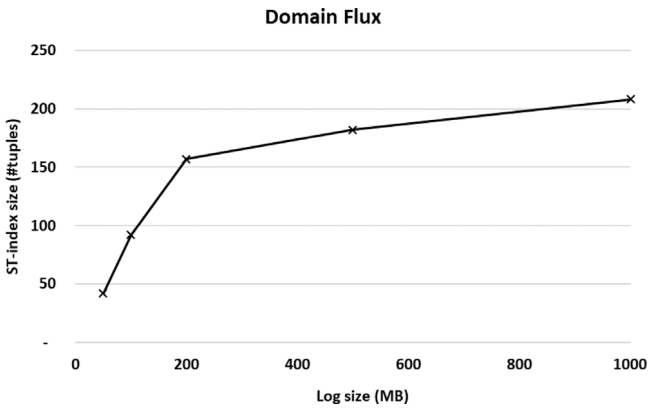


Fig. 5. Size of the ST-Index when varying log size (Domain Flux).

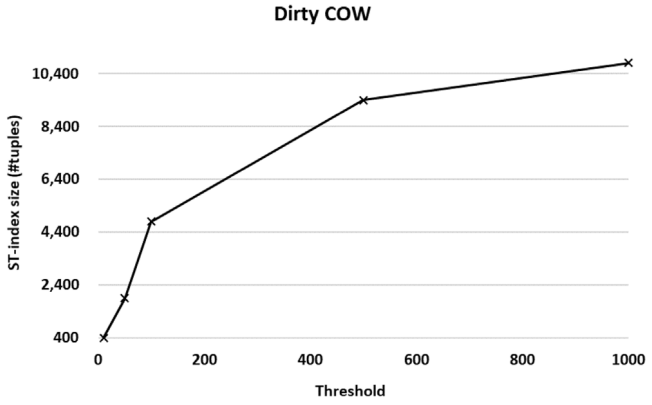


Fig. 6. Size of the ST-Index when varying the duration threshold τ (Dirty COW).

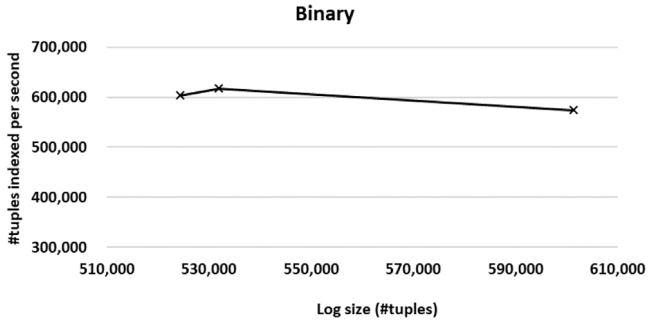


Fig. 7. Log tuple processed per second when varying log size (Binary).

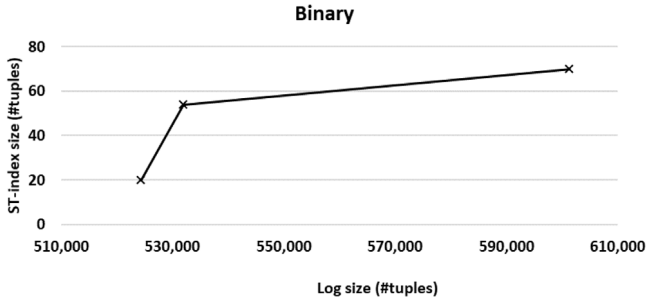


Fig. 8. Size of the ST-Index when varying log size (Binary).

represent an extendable partial sequence, the algorithm checks the various constraints (including the overall maximum duration)

Algorithm 1 Insertion

1: **procedure** INSERT(new log tuple ℓ)
2: **data:** Sequence model $M = \langle S, s_0, F, \delta, \texttt{assign}, \texttt{constr}, \tau \rangle$
3: **data:** ST-Index $\mathcal{I}_{M,L} = \langle \mathcal{P}, LS \rangle$
4: **post-condition:** $\mathcal{I}_{M,L}$ reflects the addition of ℓ to L
5: // Can ℓ be the first tuple of a target sequence?
6: **for** every $s \in S$ s.t. $(s_0, \ell.act, s) \in \delta$ **do**
7: $\theta \leftarrow \emptyset$
8: **for** every $[?X \leftarrow name(D_j)] \in \texttt{assign}(s)$ **do**
9: add $?X \mapsto \ell.name(D_j)$ to θ
10: **if** θ satisfies the conditions in Definition 3 **then**
11: add $\langle \overline{\ell}, \bot, \theta, \ell.ts, true \rangle$ to $P(s)$
12: // Can ℓ extend a partial sequence?
13: **for** every $s, s' \in S$ s.t. $(s', \ell.act, s) \in \delta$ **do**
14: **for** every $t' = \langle l', prec', \theta', t'_0, ext' \rangle \in P(s')$ s.t. $ext' = true$ **do**
15: **if** $\ell.ts - t'_0 \leq \tau$ **then**
16: $\theta \leftarrow \theta'$
17: **for** every $[?X \leftarrow name(D_j)] \in \texttt{assign}(s)$ **do**
18: add $?X \mapsto \ell.name(D_j)$ to θ
19: **if** θ satisfies the conditions in Definition 3 **then**
20: add $\langle \overline{\ell}, \overline{t'}, \theta, t'_0, s \notin F \rangle$ to $P(s)$
21: **else**
22: $ext' \leftarrow false$
23: **end procedure**

and possibly adds a corresponding row to the state tracking table (with $ext = false$ iff the new state is final).

Algorithm 1 applies a first form of pruning each time a new log tuple is indexed: the algorithm immediately identifies rows that can no longer be extended based on the information stored in the *ext* attribute of state tracking tables' rows (Line 14). In addition, a concurrent cleaning process is applied to the tables at a fixed frequency, that takes care of removing (i) every row with $ext = false$ and (ii) every row with $ext = true$ such that the difference between the most recent log tuple's timestamp and the row's t_0 is larger than τ (i.e., the partial sequence can no longer be extended or completed within the allowed maximum duration).

5.3. Implementation issues

We implemented our index and its associated maintenance algorithm in a modular framework, whose development posed several interesting challenges. In this section, we describe the implementation choices made and the resulting capabilities we obtained.

Nowadays, the majority of systems devoted to the detection of malevolent activities are bound to specific data sources. Our framework is capable of dealing with different types of logs through the use of a *preprocessing module* that extracts data from logs and then shapes them into appropriate intermediate structures that are used as logs when building the ST-Index. The preprocessing module is implemented as a set of Python classes, each built for a specific type of log. This separation between data preprocessing and indexing allows our framework to deal with many different kinds of data sources, e.g., both real time and historical data. Every time we need to deal with a new kind of log (e.g., a log generated with a new tool), we just need to add a Python class to the framework.

For instance:

- When the malevolent activities of interest are related to network activity (such as the *Fast Flux* and *Domain Flux*

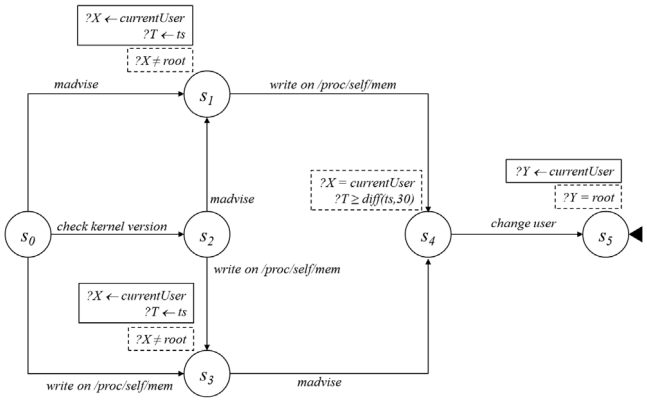


Fig. 9. Sequence model for the Dirty COW scenario.

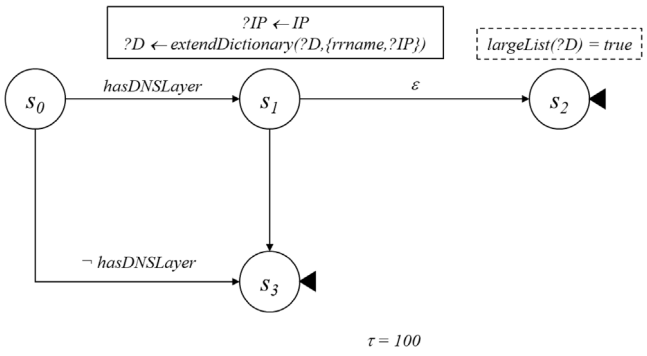


Fig. 10. Sequence model for the Fast Flux scenario.

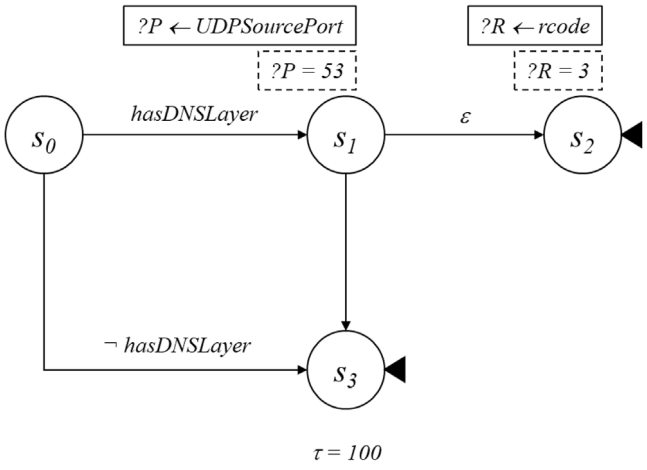


Fig. 11. Sequence model for the Domain Flux scenario.

scenarios we will describe in Section 5.4), we use `pcap` packet capture files collected by both honeypots and network sniffers (e.g., *tcpdump* and *Wireshark*). Packets can be fed to our framework either by loading historical data or by capturing packets in real time, using one of our specific Python preprocessing classes acting as a network sniffers. These functionalities are achieved thanks to the use of `scapy` Python bindings. `scapy` is a well known interactive packet manipulation tool, bundled with a lot of powerful features. The data read through `scapy` is filtered in order to extract information of interest. The tool is used also in another preprocessing class, since it also provides a useful `sniff` function. As said before, the framework can be easily extended to cover traffic captured by many other packet sniffers.

- When the malevolent activities of interest are composed of binary instruction codes (such as the malware scenario in Section 5.4), it may suffice to preprocess binary files by (i) representing commands as single instruction codes and (ii) incrementally adding timestamps to every instruction code.

The output of the preprocessing module is used in conjunction with the sequence model in order to build the ST-Index. This phase is implemented in the *analysis module*. The data structures of the ST-Index (both the state tracking tables in $\mathcal{P}$ and the label-state table *LS*) are implemented using standard Python data structures.

Finally, an *evaluation module* is used to coordinate the experimental assessment and collect the results. This module contains a *postprocessing submodule* which is devoted to the identification of correlated target sequences — for instance, in the case of our running example, this submodule chains target instances of the sequence model in order to fully reconstruct exploitations of the Dirty COW vulnerability.

5.4. Experimental evaluation

In order to evaluate the actual performance of our proposed index and algorithm, we performed two rounds of experiments. In Round 1, besides the Dirty COW scenario used as our running example (in the variant reported in the Appendix, Fig. 9), we considered two real world scenarios with malicious network activities. In Round 2, we considered a malware analysis scenario over binary files. In both rounds, we focused our attention on two main performance aspects. First, we looked at the computational overhead imposed by our indexing techniques in the activity detection process. Then, we evaluated the memory consumption of the index itself. All the experiments have been performed on a PC equipped with an Intel Core i7-5500U CPU with 8 GB RAM and Fedora 30 (kernel version 5.1.19).

Round 1 — Setting

In Round 1, we considered *Fast Flux* and *Domain Flux*, two similar DNS techniques used to hide malware delivery, in addition to the Dirty COW scenario. Fast Flux was first identified during the analysis of the *Storm* botnet [25]. In Fast Flux, a large number of IP addresses are associated with a single fully qualified domain name. The IP addresses are swapped very frequently by changing DNS records and ensuring that DNS results have a very short time-to-live, in order to force the host to re-check, in a short period of time, the validity of the IP address associated with the domain name.[7] Things went even worse in the following years with the advent of *Conficker* worm, which infected millions of computers in over 190 countries, making it the most widespread computer worm since the 2003 *Welchia* [27,28]. Since a machine infected by Conficker contacted the command-and-control server in order to download further instructions and updates, stopping the communication between infected machines and command-and-control servers was a crucial task. Conficker, however, used a *domain generation algorithm* which periodically generated a large number of domain names. The algorithm rotated domain names, thus the technique was called *Domain Flux*.[8]

As to the logs:

[7] According to [26], the Storm botnet used 2000 redundant host spread among 384 providers in more than 50 countries. This technique made it very difficult to identify a valid countermeasure, as it required to identify and take offline the command-and-control servers.

[8] The *Conficker D* variant used it to download instructions and updates daily from 500 out of 50,000 pseudorandom domains (generated every three hours) over 110 top level domains [29].

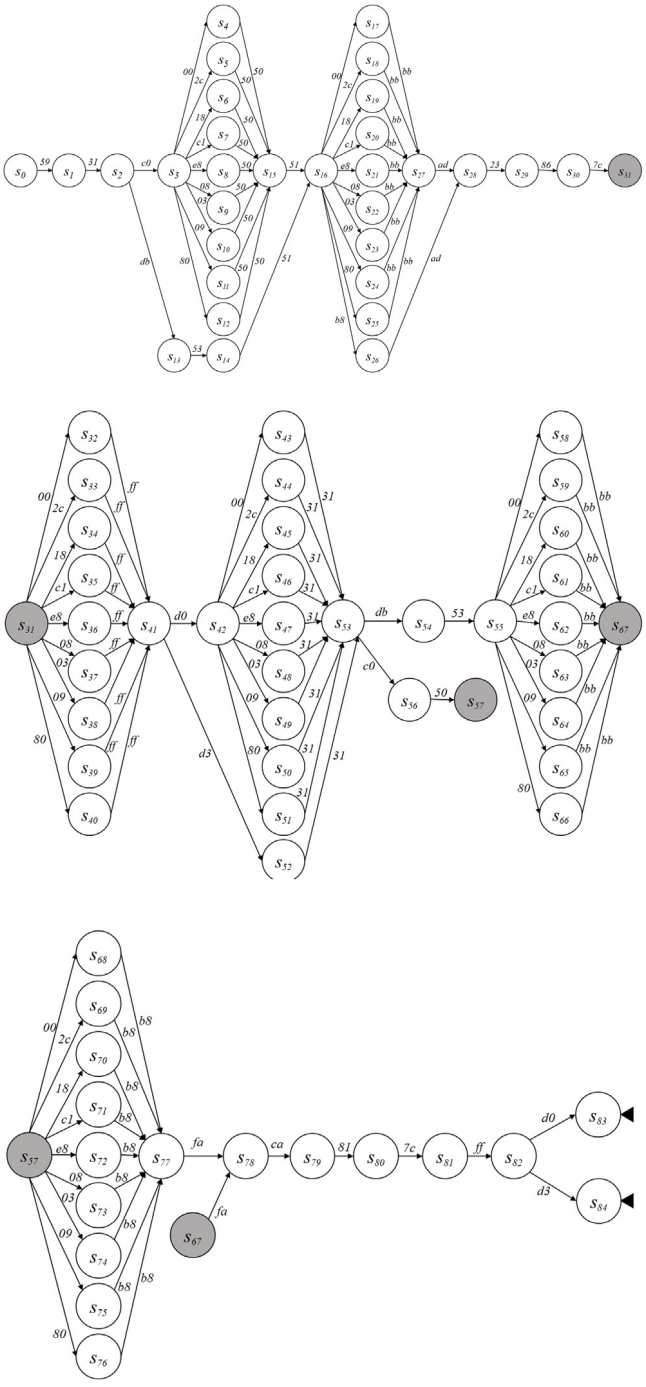


Fig. 12. Sequence model for the binary analysis scenario (gray states connect different parts of the model).

- In the case of Dirty COW, we used a logging system based on the *strace* Linux command that captures the system calls invoked by a malicious software we created ad-hoc — the software performs many different calls other than those involved in a Dirty COW attack. The log contained the timestamp, the user performing the call, the name of the call, and its parameters. We further enriched the information contained in the logs by running our logging system on many different applications, using different users, so that the results contain information related both to normal usage of the operating system and to malicious activity.

Table 2
Number of log tuples versus log size.

	Dirty COW	Fast Flux	Domain Flux
50 MB	930k	91k	93k
100 MB	1.2M	180k	182k
200 MB	3.7M	371k	368k
500 MB	7.3M	915k	920k
1000 MB	14.9M	1.9M	2.1M

- In the case of Fast Flux and Domain Flux, the information collected came from *DNS Resource Records* (DNSRR), identified by means of `scapy` Python bindings, using the `haslayer` function. We extracted the *resource record name* (rrname) and the *resource data* (rdata) fields, used by the DNS protocol to store domain name and associated IP address, respectively. This information was then stored in a Python dictionary, where the keys were the domain names and the values were the lists of IP addresses associated with the domain names. Another interesting aspect that we tackled was related to the particular implementation of the Domain Flux technique in the Conficker worm. In order to mask the command-and-control servers, Conficker used a lot of bogus domain names. Using our approach, we were able to identify Domain Flux activity and bogus domain names — to this aim, in the preprocessing Python classes, we included information gathered from the *resource code* field of the DNS packet: if this code is equal to 3, then the domain name does not exist. This check actually extended our Fast Flux model, by adding the capability to not only detect Fast Flux activity and domain names involved, but also to identify bogus ones. As our main data source, we used a live DVD ISO image provided by the *European Network and Information Security Agency* (ENISA),[9] which contains different exercises demonstrating several malicious behaviors. In order to increase the size of the logs up to 1 GB, we enriched the `pcaps` from ENISA by inserting packets coming from other network captures in random positions of the network flow, obviously after fixing the timestamps.

The sequence models corresponding to Dirty COW, Fast Flux, and Domain Flux over the logs above are reported in Appendix.

We tested our proposed framework using different log sizes and different duration thresholds — specifically, we took $|L| \in \{50\text{ MB}, 100\text{ MB}, \mathbf{200\text{ MB}}, 500\text{ MB}, 1\text{ GB}\}$ and $\tau \in \{10, 50, \mathbf{100}, 500, \infty\}$ (default values in bold).

Round 1 — Computational Overhead

Our experimental results showed that the time needed to build the ST-Index, in all of the three scenarios considered, is completely negligible when compared with the time needed to just parse the log. We also tested several additional `pcap` parsers, and in all cases their speed was basically coincident with the overall speed of our framework — the differences were always under 0.1%. This essentially tells us that the actual performance bottleneck resides in the log reading and log parsing phases, whereas the overhead posed by the indexing phase is extremely small.

Round 1 — Memory Consumption

In order to precisely characterize memory consumption, we measured the size of the ST-Index (overall number of index tuples) when varying the size of the input log and the value of the duration threshold τ. Table 2 reports the (approximate) number of log tuples for the various log sizes. Figs. 3–5 show how the size of the ST-Index varies with log sizes.

[9] http://enisa.europa.eu/activities/cert/support/exercise/live-dvd-iso-images.

The results show that, in all cases, the memory consumption due to the ST-Index is extremely small when compared to the size of the input log (always under 1/1000th of the number of log tuples). In addition, the trend is clearly sublinear — this confirms the usefulness of the pruning techniques applied by our proposed insertion algorithm. As expected, the results obtained were noticeably correlated with the kind of log analyzed.

Fig. 6 shows how the duration threshold affects the size of the ST-Index in the case of Dirty COW. Besides showing the effect of pruning (again, the trend appears sublinear), the results show that the size of the index is strongly correlated with the actual number of target sequences contained in the input log, which obviously increases with the value of the threshold.

Round 2 — Setting

In Round 2, we employed a model proposed in [19] to describe possible structures of a basic malware consisting in a shellcode that creates a user and writes a file. The model captures sequences of *binary* instruction codes (represented alphanumerically for ease of presentation). Commands are represented as single action symbols, as a result of the preprocessing phase. The full sequence model used in Round 2 is reported in Appendix.

We used 3 logs generated by injecting instances of the models (with additional action symbols not present in the model) in the binary of the PuTTY application [30]. The final sizes of the logs where around 524k tuples (with 1 instance of the malware), 531k tuples (100 instances), and 601k tuples (1000 instances) [19]. Finally, we fixed $\tau = \infty$.

Round 2 — Computational Overhead

Fig. 7 shows the throughput of the framework in terms of log tuples processed per second, when varying log size. The results showed that the number of tuples processed per second is very satisfying — it is higher than 574k tuples/s in all cases, and around 598k tuples/s on average.

Round 2 — Memory Consumption.

Fig. 8 shows the size of the ST-Index when varying the size of the input log. The results confirm the insights provided by Round 1, that is (i) the memory consumption due to the ST-Index is extremely small when compared to the size of the input log (always under 1/8000th of the number of log tuples in this case) and (ii) the overall trend is again sublinear.

6. Conclusions

In this paper we have proposed a framework to support the analysis of the very large logs usually produced by security attack prevention and detection tools. We have introduced a language for modeling the attack patterns of interest, along with a detection algorithm that is able to identify, in a given log, all possible attacks conforming to the given patterns. The theoretical properties of the language and an evaluation of the detection algorithm in real world scenarios have been discussed as well. As future work, we plan to develop variants of the algorithm that are specifically designed for parallel/distributed computing infrastructures (e.g., along the lines followed in [18]). Such kinds of infrastructures pose specific important challenges so that the extensions to our basic algorithm will eventually improve the capability of our approach to support activity detection tasks on a richer variety of logs and models.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix

Fig. 9 shows the extended sequence model of our running example based on Dirty COW.

Fig. 10 shows the sequence model used in the experimental assessment to describe Fast Flux. In the figure, (i) function *extendDictionary* adds a new value to the list associated with an existing dictionary key or creates a new (key,[value]) pair, and (ii) function *largeList* checks whether the set of dictionary values contains a list whose size exceeds a given threshold. The postprocessing submodule is employed in order to separate sequences ending in s_3 (for other analyses or archiving) from sequences ending in s_2 (possible cases of Fast Flux).

Fig. 11 shows the sequence model used in the experimental assessment to describe Domain Flux. In this case, the postprocessing submodule is employed in order to (i) correlate the number of target sequences collected, (ii) mark a situation as critical whenever the number of target sequences exceeds a given threshold, and (iii) separate sequences ending in s_3 (for other analyses or archiving) from sequences ending in s_2 (possible cases of Domain Flux).

It should be observed that both the models in Figs. 10 and 11 use ϵ-transitions, i.e., stage transitions that are triggered without an associated action. We prefer to include ϵ-transitions in the drawings for clarity of presentation — the models can easily be converted into equivalent ones without such transitions [31].

Fig. 12 shows the sequence model used in the experimental assessment to describe the malware for the binary analysis scenario. In order to take into account different forms the malevolent code can take, the model contains 5 portions where 9 different binary codes can appear.

References

[1] V. Bartos, M. Zádník, S.M. Habib, E. Vasilomanolakis, Network entity characterization and attack prediction, Future Gener. Comput. Syst. 97 (2019) 674–686.

[2] A. Qamar, A. Karim, V. Chang, Mobile malware attacks: Review, taxonomy & future directions, Future Gener. Comput. Syst. 97 (2019) 887–909.

[3] T. Yaqoob, A. Arshad, H. Abbas, M.F. Amjad, N. Shafqat, Framework for calculating return on security investment (ROSI) for security-oriented organizations, Future Gener. Comput. Syst. 95 (2019) 754–763.

[4] S. Plaga, N. Wiedermann, S.D. Antón, S. Tatschner, H.D. Schotten, T. Newe, Securing future decentralised industrial IoT infrastructures: Challenges and free open source solutions, Future Gener. Comput. Syst. 93 (2019) 596–608.

[5] O. Bouziani, H. Benaboud, A.S. Chamkar, S. Lazaar, A comparative study of open source IDSs according to their ability to detect attacks, in: Proceedings of the 2nd International Conference on Networking, Information Systems & Security, in: NISS19, ACM, 2019, pp. 51:1–51:5.

[6] A. Khraisat, I. Gondal, P. Vamplew, J. Kamruzzaman, Survey of intrusion detection systems: techniques, datasets and challenges, Cybersecurity 2 (1) 20.

[7] M. Nawrocki, M. Wählisch, T.C. Schmidt, C. Keil, J. Schönfelder, A survey on honeypot software and data analysis, 2016, arXiv.org1608.06249.

[8] M. Akiyama, T. Yagi, T. Yada, T. Mori, Y. Kadobayashi, Analyzing the ecosystem of malicious URL redirection through longitudinal observation from honeypots, Comput. Secur. 69 (2017) 155–173.

[9] C.K. Ng, L. Pan, Y. Xiang, Honeypot Frameworks and Their Applications: A New Framework, in: SpringerBriefs on Cyber Security Systems and Networks, 2018.

[10] W.M.P. van der Aalst, Process Mining - Data Science in Action, second ed., Springer, 2016.

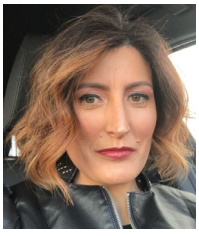
[11] J. Carmona, B.F. van Dongen, A. Solti, M. Weidlich, Conformance Checking - Relating Processes and Models, Springer, 2018.

[12] J. Munoz-Gama, Conformance Checking and Diagnosis in Process Mining - Comparing Observed and Modeled Processes, in: Lecture Notes in Business Information Processing, vol. 270, Springer, 2016.

[13] S.J.J. Leemans, D. Fahland, W.M.P. van der Aalst, Scalable process discovery and conformance checking, Softw. Syst. Model. 17 (2) (2018) 599–631.

[14] G. Greco, A. Guzzo, F. Lupia, L. Pontieri, Process discovery under precedence constraints, TKDD 9 (4) (2015) 32:1–32:39.

[15] M. Albanese, S. Jajodia, A. Pugliese, V.S. Subrahmanian, Scalable analysis of attack scenarios, in: Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12–14, 2011. Proceedings, 2011, pp. 416–433.
[16] M. Albanese, A. Pugliese, V.S. Subrahmanian, Fast activity detection: Indexing for temporal stochastic automaton-based activity models, IEEE Trans. Knowl. Data Eng. 25 (2) (2013) 360–373.
[17] A. Pugliese, A. Rullo, A. Piccolo, The AC-index: Fast online detection of correlated alerts, in: Security and Trust Management - 11th International Workshop (STM) 2015, Vienna, Austria, September 21-22, 2015, Proceedings, 2015, pp. 107–122.
[18] A. Pugliese, V.S. Subrahmanian, C. Thomas, C. Molinaro, PASS: A parallel activity-search system, IEEE Trans. Knowl. Data Eng. 26 (8) (2014) 1989–2001.
[19] A. Guzzo, A. Pugliese, A. Rullo, D. Saccà, A. Piccolo, Malevolent activity detection with hypergraph-based models, IEEE Trans. Knowl. Data Eng. 29 (5) (2017) 1115–1128.
[20] C. Molinaro, V. Moscato, A. Picariello, A. Pugliese, A. Rullo, V.S. Subrahmanian, PADUA: Parallel architecture to detect unexplained activities, ACM Trans. Internet Technol. 14 (1) (2014) 3:1–3:28.
[21] G. Cugola, A. Margara, Processing flows of information: From data stream to complex event processing, ACM Comput. Surv. 44 (3) (2012) 15:1–15:62.
[22] A. Grez, C. Riveros, M. Ugarte, Foundations of complex event processing, 2017, CoRR abs/1709.05369.
[23] A. Artikis, A. Margara, M. Ugarte, S. Vansummeren, M. Weidlich, Complex event recognition languages: Tutorial, in: Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems, DEBS 2017, Barcelona, Spain, June 19–23, 2017, ACM, 2017, pp. 7–10.
[24] D. Zimmer, R. Unland, On the semantics of complex events in active database management systems, in: International Conference on Data Engineering (ICDE) 1999, Sydney, 1999.
[25] K.J. Higgins, Attackers Hide in Fast Flux, Forbes Magazine, 2007.
[26] R. Lemos, Fast Flux Foils Bot-Net Takedown, SecurityFocus, 2007.
[27] J. Markoff, Worm Infects Millions of Computers Worldwide, The New York Times, 2009.
[28] B. Binde, R. McRee, T.J. O'Connor, Assessing Outbound Traffic to Uncover Advanced Persistent Threat, White Paper, 2011.
[29] J. Park, W32.Downadup.C Pseudo-Random Domain Name Generation, Symantec Official Blog, 2009.
[30] S. Tatham, PuTTY SSH and telnet client, 2016, http://www.putty.org. Version 0.65 for Windows.
[31] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Pearson Education Limited, 2000.



Antonella Guzzo received the Ph.D. degree in computer and systems engineering from the University of Calabria, Italy, in 2005. Currently, she is an associate professor of computer science in the DIMES Department, University of Calabria. Previously, she was a research fellow in the High Performance Computing and Networks Institute (ICAR-CNR), National Research Council, Italy. Her research interests include process mining, data mining, and knowledge representation. She is a member of the IEEE.