Abstraction-Based Malware Analysis Using Rewriting and Model Checking

Philippe Beaucamps[1], Isabelle Gnaedig[2], and Jean-Yves Marion[1]

[1] Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France
[2] Inria, Villers-lès-Nancy, F-54600, France
{Philippe.Beaucamps,Isabelle.Gnaedig,Jean-Yves.Marion}@loria.fr

Abstract. We propose a formal approach for the detection of high-level malware behaviors. Our technique uses a rewriting-based abstraction mechanism, producing abstracted forms of program traces, independent of the program implementation. It then allows us to handle similar behaviors in a generic way and thus to be robust with respect to variants. These behaviors, defined as combinations of patterns given in a signature, are detected by model-checking on the high-level representation of the program. We work on unbounded sets of traces, which makes our technique useful not only for dynamic analysis, considering one trace at a time, but also for static analysis, considering a set of traces inferred from a control flow graph. Abstracting traces with rewriting systems on first order terms with variables allows us in particular to model dataflow and to detect information leak.

Keywords: Malware, behavioral detection, behavior abstraction, trace, term rewriting, model checking, first-order temporal logic, finite state automaton, formal language.

1 Introduction

Behavior analysis was introduced by Cohen's seminal work [1] to detect malware and in particular unknown malware. In general, a behavior is described by a sequence of system calls and recognition uses the formalism of finite state automata [2,3,4,5]. New approaches have been proposed recently. In [6,7], malicious behaviors are specified by temporal logic formulas with parameters and detection is carried out by model-checking. However, these approaches are tightly dependent on the way malicious actions are realized: using any other system facility to realize an action allows a malware to go undetected. This has motivated yet another approach where a malicious behavior is specified as a combination of high-level actions, in order to be independent from the way these actions are realized and to only consider their effect on a system. In [8] and in [9], a captured execution trace is transformed into a higher-level representation capturing its semantic meaning, i.e., the trace is first abstracted before being compared to a malicious behavior. In [10], the authors propose to use attribute automata, at the price of an exponential time complexity detection. These dynamic abstraction-based

S. Foresti, M. Yung, and F. Martinelli (Eds.): ESORICS 2012, LNCS 7459, pp. 806–823, 2012.

approaches, though they can detect unknown viruses whose execution traces exhibit known malicious behaviors, only deal with a single execution trace.

In this paper, we propose a formal approach for high-level behavior analysis, with the following features. Underpinned by language theory, term rewriting and first-order temporal logic, it allows us to determine whether a program exhibits a high-level behavior. Detection is achieved in two steps. First, traces of the program are abstracted in order to reveal the sequences of high-level functionalities they realize. Then, abstracted traces are compared with the behavior formula, using usual model-checking techniques. Functionalities have parameters representing the manipulated data, so our formalism is adapted to the protection against generic threats like the leak of sensitive information.

Our goal here is not to provide a ready-made software to detect behaviors, but to propose a formal framework emphasizing fundamental detection mechanisms, which are independent of implementation-based solutions.

Our approach has two main characteristics. First, we work on an unbounded set of traces representing the behavior of a program, in order to consider a more complete representation of the program than with a single trace. To deal with the infinity of the set of traces, we restrict to regular sets and safely approximate the set of abstract traces, so that we detect in linear time whether a program exhibits a given behavior. Second, we work on abstract forms of traces, in order to only keep the essence of the functions performed by the program, to be independent of their possible implementations and to be generic with respect to behavior mutations. Behavior components are abstracted in program traces, by identifying known functionalities and marking them by inserting abstract functionality symbols.

By working on sets of traces, which may consist of a single trace as well as of an unbounded number of traces, our approach may be used not only for classical, dynamic behavior analysis, but also for static behavior analysis i.e., behavior analysis in a static analysis setting.

Static behavior analysis by abstraction is more challenging than its dynamic counterpart because, precisely, this approach needs to abstract a program behavior potentially representing an infinite set of execution traces. The construction of an exhaustive representation of a program behavior is an intractable problem in general: in particular, a program flow may not be easily followed due to indirect jumps, and a program may use complex code protection, for instance by dynamically modifying its code or by using obfuscation. Self modification is usually tackled by emulating the program long enough to deactivate most code protections. Indirect jumps and obfuscation are usually handled by abstract interpretation [11,12] or symbolic execution [13].

Static behavior analysis has many advantages and applications. First, it allows us to analyze the behavior of a program in a more exhaustive way, as it analyzes the unbounded set of the program execution traces, or an approximation of it. Second, static behavior analysis can complement classical, dynamic, behavior analysis with an analysis of the future behavior, to prevent damages when some critical point is reached in an execution.

An interesting application of static behavior analysis is the audit of programs in high-level technologies, like mobile applications, browser extensions, web page scripts, .NET or Java programs. Auditing these programs is complex and mostly manual, resulting in highly publicized infections [14,15]. In this context, static analysis can provide an appropriate help, because it is usually easier than for usual programs, especially when additionally enforcing a security policy (e.g. prohibiting self-modification [16]) or when enforcing strict development guidelines (e.g. for iPhone applications).

To our knowledge, the use of behavior abstraction on top of static behavior analysis has not been investigated so far. As our detection mechanism relies on satisfaction of temporal logic formulas, it is akin to model checking [17], for which there already exist numerous frameworks and tools [18,19,20]. The specificity of our approach, however, is that, rather than being applied on the set of program traces, verification is applied on the set of abstract forms of these traces, which is not computable in general. Accordingly, we identify a property of practical high-level behaviors allowing us to approximate this set, in a sound and complete way with respect to detection, and then to apply classical verification techniques.

Our abstraction framework can be used in two scenarios:

- *Detection of given behaviors*: signatures of given high-level behaviors are expressed in terms of abstract functionalities. Given some program, we then assess whether one of its execution traces exhibits a sequence of known functionalities, in a way specific to one of the given behaviors. This can be applied to detection of suspicious behaviors. Although detection of such suspicious behaviors may not suffice to label a program as malicious, it can be used to supplement existing detection techniques with additional decision criteria.
- *Analysis of programs*: abstraction provides a simple and high-level representation of a program behavior, which is more suitable than the original traces for manual analysis, or for analysis of behavior similarity with known behaviors, etc. For instance, it could be used to detect not necessarily harmful behaviors, in order to get a basic understanding of the program and to further investigate if deemed necessary. It could also be used to automatically discover sequences of high-level functionalities and their dataflow dependencies, exhibited by a program.

Previous Work. In [21], we already proposed to abstract program sets of traces with respect to behavior patterns, for detection and analysis. We tested our approach on samples of malicious programs collected using a honeypot[1] and identified using Kaspersky Antivirus. These samples belonged to known malware families, like Allaple, Virut, Agent, Rbot, Afcore and Mimail. Most of them were successfully matched to our malware database.

But patterns were defined by string rewriting systems, which did not allow the actions composing a trace to have parameters, precluding dataflow analysis. Moreover, abstraction rules replaced identified patterns by abstraction symbols

[1] The honeypot of the Loria's High Security Lab: `http://lhs.loria.fr`

in the original trace, precluding a further detection of patterns interleaved with the rewritten ones.

The formalism proposed in this paper addresses both issues: first, we handle interleaved patterns by keeping the identified patterns when abstracting them. Second, we extend the rewriting framework to express data constraints on action parameters by using term rewriting systems. An important consequence is that, unlike in [21], using the dataflow, we can detect information leaks in order to prevent unauthorized disclosure or modifications of information.

2 Background

Term Algebras. Let $S = \{Trace, Action, Data\}$ be a set of sorts, $\mathcal{F} = \mathcal{F}_{\mathrm{t}} \cup \mathcal{F}_{\mathrm{a}} \cup \mathcal{F}_{\mathrm{d}}$ be a finite S-sorted signature, where $\mathcal{F}_{\mathrm{t}}$, $\mathcal{F}_{\mathrm{a}}$, $\mathcal{F}_{\mathrm{d}}$ are mutually distinct and:

- $\mathcal{F}_{\mathrm{t}} = \{\epsilon, \cdot\}$ is the set of the trace constructors, where $\epsilon :\rightarrow Trace$ denotes the empty trace, . has profile $Data\ Trace \rightarrow Trace$;
- $\mathcal{F}_{\mathrm{a}}$ is a set of function symbols or constants, with profile $Data^n \rightarrow Action$, $n \in \mathbb{N}$, describing actions;
- $\mathcal{F}_{\mathrm{d}}$ is a set of data constructors, with profile $\rightarrow Data$ or $Data^n \rightarrow Data$, $n \in \mathbb{N}$.

Let $\mathbb{N}_+^*$ be the set of finite strings of positive natural numbers, called *positions*. The empty string is denoted by λ, and $u \leq v$ means that u is prefix of v. Let X be a set of S-sorted variables. A S-sorted *term* over $(\mathcal{F}, X)$ is a partial function $t : \mathbb{N}_+^* \rightarrow \mathcal{F} \cup X$, such that the domain of definition of t, denoted by $\mathcal{P}os(t)$, is finite and satisfies, for $w \in \mathbb{N}_+^*$ and $i \in \mathbb{N}$: (1) $wi \in \mathcal{P}os(t) \Rightarrow w \in \mathcal{P}os(t)$, (2) $w \in \mathcal{P}os(t) \Rightarrow t(w) \in \mathcal{F} \cup X$. $\mathcal{P}os(t)$ is called the set of positions of t. We denote by $T(\mathcal{F}, X)$ (resp. $T(\mathcal{F})$) the set of S-sorted terms over $(\mathcal{F}, X)$ (resp. the set of finite ground terms over $\mathcal{F}$). For any sort $s \in S$, and any of the above sets of terms T we denote by T_s *the restriction* of T to terms of sort s and by X_s *the subset of variables* of X of sort s. For a term t with $p \in \mathcal{P}os(t)$, we denote by $t|_p$ the subterm of t at position p. We denote by $t[t']_p$ the term obtained by replacing by t' the *subterm* at position p in t. We use the abbreviated notation $\overline{x}$ for variables $x_1, \ldots, x_n$. So $\overline{x} \in X$ stands for $x_1, \ldots, x_n \in X$, and if $f \in \mathcal{F}$ is a symbol of arity $n \in \mathbb{N}$, we denote by $f(\overline{x})$ the term $f(x_1, \ldots, x_n)$.

The elements of $T_{\mathrm{Trace}}(\mathcal{F})$ are called *traces*, the elements of $T_{\mathrm{Action}}(\mathcal{F})$ are called *actions*. We distinguish the sort Action from the sort Trace but, for a sake of readability, we may denote by a the trace $\cdot(a, \epsilon)$, for some action a. Similarly, we use the $\cdot$ symbol with infix notation and right associativity, and ϵ is understood when the context is unambiguous. For instance, if a, b, c are actions, $a \cdot b \cdot c$ denotes the trace $\cdot(a, \cdot(b, \cdot(c, \epsilon)))$.

We partition $\mathcal{F}_{\mathrm{a}}$ in a set Σ of symbols, denoting concrete program-level actions, and a set Γ, denoting abstract actions identifying abstracted functionalities. To construct purely concrete (resp. abstract) terms, we use $\mathcal{F}_\Sigma = \mathcal{F} \setminus \Gamma$ (resp. $\mathcal{F}_\Gamma = \mathcal{F} \setminus \Sigma$). The *projection* $t|_{\Sigma'}$, also denoted $\pi_{\Sigma'}(t)$, of a trace t on an alphabet $\Sigma' \subseteq \mathcal{F}_{\mathrm{a}}$ corresponds to keeping in a trace only actions from Σ'. If X is

a set of variables of sort *Data*, we define the projection on an alphabet $\Sigma' \subseteq \mathcal{F}_{\mathrm{a}}$ of a term $t \in T_{\mathrm{Trace}}(\mathcal{F}, X)$, denoted by $\pi_{\Sigma'}(t)$ or, equivalently, by $t|_{\Sigma'}$, in the following way:

$$\pi_{\Sigma'}(\epsilon) = \epsilon$$
$$\pi_{\Sigma'}(b \cdot u) = \begin{cases} b \cdot \pi_{\Sigma'}(u) & \text{if } b \in T_{\mathrm{Action}}(\mathcal{F}_{\Sigma'}, X) \\ \pi_{\Sigma'}(u) & \text{otherwise} \end{cases}$$

with $b \in T_{\mathrm{Action}}(\mathcal{F}, X)$ and $u \in T_{\mathrm{Trace}}(\mathcal{F}, X)$. The projection is naturally extended to sets of traces.

We define in a natural way the *concatenation* $t \cdot t'$ of two traces t and t'. The concatenation of two terms t and t' of $T_{\mathrm{Trace}}(\mathcal{F}, X)$, where X is a set of S-sorted variables and $t \notin X$, is denoted by $t \cdot t' \in T_{\mathrm{Trace}}(\mathcal{F}, X)$ and defined by $t \cdot t' = t[t']_p$, where p is the position of ϵ in t, i.e., $t|_p = \epsilon$. Projection and concatenation are naturally extended to sets of terms of sort Trace. We also extend concatenation to $2^{T_{\mathrm{Trace}}(\mathcal{F},X)} \times 2^{T_{\mathrm{Trace}}(\mathcal{F},X)}$ with $L \cdot L' = \{t \cdot t' \mid t \in L, t' \in L'\}$ and to $2^{T_{\mathrm{Trace}}(\mathcal{F},X)} \times T_{\mathrm{Action}}(\mathcal{F}, X)$ with $L \cdot a = L \cdot \{a \cdot \epsilon\}$.

Substitutions are defined as usual. A *ground substitution* on a finite set X of S-sorted variables is a mapping $\sigma : X \to T(\mathcal{F})$ such that: $\forall s \in S, \forall x \in X_s$, $\sigma(x) \in T_s(\mathcal{F})$. σ can be naturally extended to a mapping $T(\mathcal{F}, X) \to T(\mathcal{F})$ in such a way that:

$$\forall f(t_1, \ldots, t_n) \in T(\mathcal{F}, X),$$
$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)) \quad .$$

By convention, we denote by $t\sigma$ or by $\sigma(t)$ the application of a substitution σ to a term $t \in T(\mathcal{F}, X)$ and by $L\sigma$ the application of σ to a set of terms $L \subseteq T(\mathcal{F}, X)$. The *set of ground substitutions* over X is denoted by $Subst_{\mathrm{X}}$.

Program Behavior. The representation of a program is chosen to be its set of traces. When executing a program, the captured data is represented on the alphabets Σ, denoting the concrete actions, and $\mathcal{F}_{\mathrm{d}}$, describing the data. In this paper, we consider that the captured data is the library calls along with their arguments. Σ therefore represents the finite set of library calls, while terms built on $\mathcal{F}_{\mathrm{d}}$ identify the arguments and the return values of these calls. A *program execution trace* then consists of a sequence of library calls and is defined by a term of $T_{\mathrm{Trace}}(\mathcal{F}_\Sigma)$. A *program behavior* is defined by the set of its execution traces, that is a possibly infinite subset of $T_{\mathrm{Trace}}(\mathcal{F}_\Sigma)$. For instance, the term $fopen(1,2) \cdot fwrite(1,3)$ represents the execution trace of a file open call $fopen(1,2)$ followed by a file write call $fwrite(1,3)$, where $1 \in \mathcal{F}_{\mathrm{d}}$ identifies the file handle returned by $fopen$, $2 \in \mathcal{F}_{\mathrm{d}}$ identifies the file path and $3 \in \mathcal{F}_{\mathrm{d}}$ identifies the written data.

First-Order Linear Temporal Logic (FOLTL). We consider the First-Order Logic (FOLTL) defined in [17], without the equality predicate, where the set of atomic predicates AP is a set of terms with variables in a set X. FOLTL is an extension of the LTL Logic (see also [17]) such that:

- If φ is an LTL formula, then φ is an FOLTL formula;
- If φ is an FOLTL formula and $Y \subseteq X$ is a set of variables, then: $\exists Y.\varphi$ and $\forall Y.\varphi$ are FOLTL formulas, where as usual: $\forall Y.\varphi \equiv \neg\exists Y.\neg\varphi$.

Notation $\varphi_1 \odot \varphi_2$ stands for $\varphi_1 \wedge \mathbf{X}(\top \mathbf{U} \varphi_2)$.

We say that a FOLTL formula is *closed* when it has no free variable, i.e., every variable is bound by a quantifier.

Let $Y \subseteq X$ be a set of variables of sort *Data* and $\sigma \in Subst_Y$ be a ground substitution over Y. The *application of* σ to an FOLTL formula φ is naturally defined by the formula $\varphi\sigma$ where any free variable x in φ which is in Y has been replaced by its value $\sigma(x)$.

A formula φ is *satisfied* on infinite sequences of sets of ground instances of atomic predicates, denoted by $\xi = (\xi_0, \xi_1, \ldots)$. $\xi \models \varphi$ (ξ satisfies φ) is defined in the same way as for the LTL logic, with the additional rule: $\xi \models \exists Y.\varphi$ iff there exists $\sigma \in Subst_Y$ such that $\xi \models \varphi\sigma$.

In our context, a formula is satisfied over traces of $T_{\text{Trace}}(\mathcal{F})$ identified with sequences of singleton sets of atomic predicates. A finite trace $t = a_0 \cdots a_n$ is identified with the infinite sequence of sets of atomic predicates $\xi_t = (\{a_0\}, \ldots, \{a_n\}, \{\}, \{\}, \ldots)$, and t satisfies φ, denoted by $t \models \varphi$, iff $\xi_t \models \varphi$.

We consider two distinct instances of this logic, depending on the fact that we consider concrete traces or abstract traces. We denote by FOLTL_Σ the FOLTL logic, where the set of atomic predicates is $AP_\Sigma = T_{\text{Action}}(\mathcal{F}_\Sigma, X)$ and ξ is in $\left(2^{T_{\text{Action}}(\mathcal{F}_\Sigma)}\right)^\omega$. We denote by FOLTL_Γ the FOLTL logic, where the set of atomic predicates is $AP_\Gamma = T_{\text{Action}}(\mathcal{F}_\Gamma, X)$ and ξ is in $\left(2^{T_{\text{Action}}(\mathcal{F}_\Gamma)}\right)^\omega$.

Note that in practice, to express behaviors, we only use FOLTL formulas that are negations of safety properties. We do not use properties with liveness aspects, which would note make sense on finite traces. Using FOLTL on finite traces allows us a correct balance between behavior expressivity and decidability.

Tree automata and tree transducers are defined as usual [22].

3 Behavior Patterns

The problem under study can be formalized in the following way. First, using FOLTL formulas, we define a set of behavior patterns, where each pattern represents a (possibly infinite) set of terms from $T_{\text{Trace}}(\mathcal{F}_\Sigma)$. Second, we need to define a terminating abstraction relation R allowing to schematize a trace by abstracting occurrences of the behavior patterns in that trace. Finally, given some program p coming with an infinite set of traces L (static analysis scenario, for instance by using the control flow graph, see our previous work [21] and [23,24]), we formulate the *detection problem* in the following way. Let $L{\downarrow_R}$ be the set of normal forms of traces of L for R i.e., the set of abstracted traces of L, using R. Given an abstract behavior M defined by an FOLTL formula φ, does there exist a trace t in $L{\downarrow_R}$ such that $t \models \varphi$? Our goal is then to find an effective and efficient method solving this problem.

A behavior pattern describes a functionality we want to recognize in a program trace, like writing to system files, sending a mail or pinging a remote host. Such a

functionality can be realized in different ways, depending on which system calls, library calls or programming languages it uses.

We describe a functionality by an FOLTL formula, such that traces satisfying this formula are traces carrying out the functionality.

Example 1. Let us consider the functionality of sending a ping. One way of realizing it consists in calling the *socket* function with the parameter `IPPROTO_ICMP` describing the network protocol and, then, calling the *sendto* function with the parameter `ICMP_ECHOREQ` describing the data to be sent. Between these two calls, the socket should not be freed. This is described by the FOLTL formula: $\varphi_1 = \exists x, y.\, socket\,(x, \alpha) \wedge (\neg closesocket\,(x)\ \mathbf{U}\ sendto\,(x, \beta, y))$, where the first parameter of *socket* is the created socket and the second parameter is the network protocol, the first parameter of *sendto* is the used socket, the second parameter is the sent data and the third one is the target, the unique parameter of *closesocket* is the freed socket and constants α and β in $\mathcal{F}_{\mathrm{d}}$ identify the above parameters `IPPROTO_ICMP` and `ICMP_ECHOREQ`.

A ping may also be realized using the function *IcmpSendEcho*, whose parameter represents the ping target. This corresponds to the FOLTL formula: $\varphi_2 = \exists x.\, IcmpSendEcho\,(x)$.

Hence, the ping functionality may be described by the FOLTL formula: $\varphi_{\mathrm{ping}} = \varphi_1 \vee \varphi_2$.

We then define a behavior pattern as the set of traces carrying out its functionality i.e., satisfying the formula describing the functionality.

Definition 1. *A* behavior pattern *is a set of traces* $B \subseteq T_{\mathrm{Trace}}\,(\mathcal{F}_\Sigma)$ *satisfying a closed* FOLTL_Σ *formula* φ*:* $B = \{t \in T_{\mathrm{Trace}}\,(\mathcal{F}_\Sigma) \mid t \models \varphi\}$.

4 Trace Abstraction

As said before, our goal is to be able to detect, in a given set of traces, some predefined behavior composed of combinations of high-level functionalities. For this, we associate to each behavior pattern an abstract symbol λ taken in the alphabet Γ, called abstraction symbol. An abstract behavior is then defined by combinations of abstraction symbols associated to behavior patterns, using an FOLTL formula φ on $AP_\Gamma = T_{\mathrm{Action}}\,(\mathcal{F}_\Gamma, X)$ instead of $AP_\Sigma = T_{\mathrm{Action}}\,(\mathcal{F}_\Sigma, X)$.

Definition 2. *An* abstract behavior *is a set of traces* $M \subseteq T_{\mathrm{Trace}}\,(\mathcal{F}_\Gamma)$ *satisfying a closed* FOLTL_Γ *formula* φ_M*:* $M = \{t \in T_{\mathrm{Trace}}\,(\mathcal{F}_\Gamma) \mid t \models \varphi_M\}$. *When* M *is defined by a formula* φ_M*, we write:* $M := \varphi_M$.

Example 2. The abstract behavior of sending a ping to a remote host can then be trivially defined by the formula: $\varphi_M = \exists x.\, \mathbf{F}\, \lambda_{\mathrm{ping}}\,(x)$.

In the following, for the sake of simplicity, the initial $\mathbf{F}$ operator is implicit in definitions of abstract behaviors.

Now, let L be the set of program traces we want to analyze. To compare these traces to the given abstract behavior, we have to consider the behavior pattern occurrences they may contain, at the abstract level. For this, we define an abstraction relation R, which marks such occurrences in traces by inserting an abstraction symbol λ_B when an occurrence of the behavior pattern B is identified.

From now on, if a behavior pattern is defined using an FOLTL formula φ and associated to an abstraction symbol λ, we may denote it $\lambda := \varphi$.

The abstraction symbol can have parameters corresponding to those used by the behavior pattern. This allows us to express dataflow constraints in a signature. For instance, the abstraction symbol for the ping behavior pattern can take a parameter denoting the ping target. A signature for a denial of service could then be defined, for example, as a sequence of 100 pings with the same target.

Example 3. The ping behavior pattern in Example 1 is abstracted in traces by inserting the λ_{ping} symbol after the *send* action or after the *IcmpSendEcho* action. Then, the trace $socket(1, \alpha) \cdot gethostbyname(2) \cdot sendto(1, \beta, 3) \cdot closesocket(1)$ can be abstracted into the trace $socket(1, \alpha) \cdot gethostbyname(2) \cdot sendto(1, \beta, 3) \cdot \lambda_{\text{ping}}(3) \cdot closesocket(1)$.

Thus, abstraction of a trace reveals abstract behavior pattern combinations, which may constitute the abstract behavior to be observed. We now formally define the abstraction relation.

As said above, abstracting a trace with respect to some behavior pattern amounts to transforming it when it contains an occurrence of the behavior pattern, by inserting a symbol of Γ in the trace. This symbol is inserted at the position after which the behavior pattern functionality has been performed. This position is the most logical one to stick to the trace semantics. Furthermore, when behavior patterns appear interleaved, this position allows us to define the order in which their functionalities are realized (see the full version of the paper for an example [25]).

As said in the introduction, rather than replace behavior pattern occurrences with abstraction symbols, we preserve them in order to properly handle interleaved behavior patterns occurrences. Now, let us consider the following example.

Example 4. Abstraction of the ping in Example 3 is realized by rewriting using the rule $A_1(x, y) \cdot B_1(x, y) \rightarrow A_1(x, y) \cdot \lambda(y) \cdot B_1(x, y)$, where $A_1(x, y) = socket(x, \alpha) \cdot (T_{\text{Trace}}(\mathcal{F}_\Sigma) \setminus (T_{\text{Trace}}(\mathcal{F}_\Sigma) \cdot closesocket(x) \cdot T_{\text{Trace}}(\mathcal{F}_\Sigma))) \cdot sendto(x, \beta, y)$ and $B_1(x, y) = \{\epsilon\}$, and the rule $A_2(x) \cdot B_2(x) \rightarrow A_2(x) \cdot \lambda(x) \cdot B_2(x)$, where $A_2(x) = \{IcmpSendEcho(x)\}$ and $B_2(x) = \{\epsilon\}$.

As a behavior pattern is a set of possible traces realizing a given functionality, we define the abstraction relation by decomposing the behavior pattern into a finite union of concatenations of sets $A_i(X)$ and $B_i(X)$ such that traces in $A_i(X)$ end with the action effectively performing the behavior pattern functionality. These sets $A_i(X)$ and $B_i(X)$ are composed of concrete traces only, since abstract

actions that may appear in a partially rewritten trace should not impact the abstraction of an occurrence of the behavior pattern.

Definition 3. *Let $\lambda \in \Gamma$ be an abstraction symbol, X be a set of variables of sort Data, $\overline{x}$ be a sequence of variables in X. An* abstraction system *on $T_{\text{Trace}}(\mathcal{F}, X)$ is a finite set of rewrite rules of the form: $A_i(X) \cdot B_i(X) \to A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ where the sets $A_i(X)$ and $B_i(X)$ are sets of concrete traces of $T_{\text{Trace}}(\mathcal{F}_\Sigma, X)$.*

Dealing with sets as left(right)-hand sides of rules may seem to be heavy. In fact, this allows us to recognize not only finitely enumerated patterns, but patterns from languages i.e., patterns among possibly infinite sets of behaviors.

The system of rewrite rules we use generates a reduction relation on $T_{\text{Trace}}(\mathcal{F})$ such that filtering works on traces projected on Σ.

Definition 4. *The* reduction relation *on $T_{\text{Trace}}(\mathcal{F})$ generated by a system of n rewrite rules $A_i(X) \cdot B_i(X) \to A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ is the rewriting relation $\to_{\mathcal{R}}$ such that, for all $t, t' \in T_{\text{Trace}}(\mathcal{F}), t \to_{\mathcal{R}} t'$ iff:*

$$\exists \sigma \in Subst_{\text{X}}, \exists p \in \mathcal{P}os(t), \exists i \in [1..n],$$
$$\exists a \in T_{\text{Trace}}(\mathcal{F}) \cdot T_{\text{Action}}(\mathcal{F}_\Sigma), \exists b, u \in T_{\text{Trace}}(\mathcal{F}),$$
$$a|_\Sigma \in A_i(X)\sigma,\ b|_\Sigma \in B_i(X)\sigma,\ t|_p = a \cdot b \cdot u$$
$$\text{and } t' = t[a \cdot \lambda(\overline{x})\sigma \cdot b \cdot u]_p \quad .$$

An abstraction relation with respect to a given behavior pattern is thus the reduction relation of an abstraction system, where left members of the rules cover the set of the traces realizing the behavior pattern functionality.

Definition 5. *Let B be a behavior pattern associated with an abstraction symbol $\lambda \in \Gamma$. Let X be a set of variables of sort Data. An* abstraction relation *w.r.t. this behavior pattern is the reduction relation on $T_{\text{Trace}}(\mathcal{F}_\Sigma)$ generated by an abstraction system composed of n rules $A_i(X) \cdot B_i(X) \to A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ verifying:*

$$B = \bigcup_{i \in [1..n]} \bigcup_{\sigma \in Subst_{\text{X}}} (A_i(X) \cdot B_i(X))\sigma \quad .$$

Finally, we generalize the definition of abstraction to a set of behavior patterns.

Definition 6. *Let C be a finite set of behavior patterns. An* abstraction relation *w.r.t C is the union of the abstraction relations w.r.t. the elements of C.*

As we will see later, for R to be realizable by a tree transducer, the abstraction relation R has to be terminating. However, even with a finite set of traces, abstraction does not terminate in general, since the same occurrence of a pattern can be abstracted an unbounded number of times. So we require that the same abstract action is not inserted twice after the same concrete action. In other words, if $t = t_1 \cdot t_2$ is abstracted into $t' = t_1 \cdot \alpha \cdot t_2$, where α is the inserted abstract action, then if t_2 starts with a sequence of abstract actions, α does not appear in

this sequence. Formally, we require that: $\forall t_1, t_2 \in T_{\text{Trace}}(\mathcal{F}), \forall \alpha \in T_{\text{Action}}(\mathcal{F}_\Gamma)$, if $t_1 \cdot t_2 \to_R t_1 \cdot \alpha \cdot t_2$, then $\nexists u \in T_{\text{Trace}}(\mathcal{F}_\Gamma), \nexists u' \in T_{\text{Trace}}(\mathcal{F}), t_2 = u \cdot \alpha \cdot u'$.

Using the above condition, supposed to be verified from now on, a behavior pattern occurrence can only be abstracted once. Furthermore, abstraction does not create new abstraction opportunities so the relation R is clearly terminating.

Remark 1. A terminating abstraction relation with respect to a set of behavior patterns is not confluent in general. We could adapt the definition of the abstraction relation to make it confluent, for instance by defining an order on the set $T_{\text{Action}}(\mathcal{F}_\Gamma)$. However, as already mentioned, detection works on normal forms. So having several normal forms for a trace does not compromise its mechanism.

In practice, a behavior pattern is regular, along with the set of instances of right-hand sides of its abstraction rules. We show that this is sufficient, with termination of the set of rules, to ensure that the abstraction relation is realizable by a tree transducer, in other words that it is a rational tree transduction. The tree transducer formalism is chosen for its interesting formal (closure by union, composition, preservation of regularity) and computational properties. When $T_{\text{Action}}(\mathcal{F})$ is finite, we can state the following result.

Theorem 1. *Let B be a behavior pattern and R be a terminating abstraction relation w.r.t. B defined by an abstraction system whose set of instances of right-hand sides of rules is recognized by a tree automaton A_R. Then R and R^{-1} are rational and, for any tree automaton A recognizing a trace language L, $R(L)$ is recognized by a tree automaton of size $O(|A| \cdot |A_R|)$.*

5 Detection Problem

Then the detection problem can be formalized as follows.

Definition 7. *A set of traces $L \subseteq T_{\text{Trace}}(\mathcal{F}_\Sigma)$* exhibits an abstract behavior *M defined by a formula φ_M, denoted by $L \Cap M$, iff: $\exists t \in L{\downarrow_R}|_\Gamma, t \models \varphi_M$.*

When L is restricted to a single trace, or to a finite set of traces, like in dynamic analysis, the set $L{\downarrow_R}$ of normal forms of traces i.e., the set of traces that cannot be rewritten anymore with R, is computable since the rewrite system R is terminating. Moreover, as FOLTL quantification is performed over variables in the domain $T_{\text{Data}}(\mathcal{F})$, FOLTL verification is decidable when $T_{\text{Data}}(\mathcal{F})$ is finite. So in this case, it can be decided whether L exhibits M.

For an infinite set of finite traces L however, the computation of $L{\downarrow_R}$ often relies on the computation of the set of descendants $R^*(L)$ of L i.e., the set of all terms that can be rewritten from terms of L. But $R^*(L)$ is computable only for some classes of rewrite systems [26] and when L is regular. Unfortunately, the rewrite systems which implement the abstraction relations and which are described in Sect. 4 do not belong to any of these classes. Hence, we cannot rely on the construction of $L{\downarrow_R}$ to decide whether L exhibits M.

Nevertheless, we will see that, for behaviors considered in practice, a partial abstraction of the set of traces is sufficient i.e., computing the set of normal forms is unnecessary. We therefore propose a detection algorithm relying on a safe approximation of the set of abstract traces. This approximation must be chosen carefully. For instance, it cannot consist in computing, for some n, the set $R^{\leq n}(L)$ of descendants of L until order n, as shown by the following example.

Example 5. Let $\lambda_1 := a$, $\lambda_2 := b$, $\lambda_3 := c$ be three behavior patterns associated to abstraction relations inserting the abstraction symbol after a, b and c respectively. Let $M := \lambda_1 \wedge (\neg\lambda_2 \,\mathbf{U}\, \lambda_3)$ be an abstract behavior. Assume there exists a bound n such that $L{\downarrow_R}$ may be approximated by $R^{\leq n}(L)$ in Definition 7. The trace $t = a^{n-1} \cdot b \cdot c \cdot d$ does not exhibit the behavior M. Yet the trace $t' = (a \cdot \lambda_1)^{n-1} \cdot b \cdot c \cdot \lambda_3 \cdot d$ is in $R^{\leq n}(\{t\})$ and its projection on Γ is in M, so we would wrongly infer that t exhibits M.

The problem comes from the fact that $R^{\leq n}(L)$ contains contradictory traces compromising detection i.e., traces seemingly exhibiting an abstract behavior though a few additional abstraction steps would make them leave the signature.

Consequently, we want to exclude traces unreliably realizing the abstract behavior in $R^{\leq n}(L)$, while not having to reach normal forms. In fact, we identify a fundamental property we call (m, n)-completeness, verified by abstract behaviors in practice in the field of malware detection. This property states that, for a program to exhibit an abstract behavior, a necessary and sufficient condition is the following: there exists a partially abstracted trace, abstracted in at most m abstraction steps, realizing the behavior and whose descendants until the order n still realize it.

Definition 8. *Let M be an abstract behavior defined by a formula φ_M and m and n be positive numbers. M has the property of* (m, n)-completeness *iff for any set of traces $L \subseteq T_{\mathrm{Trace}}(\mathcal{F}_\Sigma)$:*

$$L \Subset M \;\Leftrightarrow\; \exists t' \in R^{\leq m}(L), \forall t'' \in R^{\leq n}(t')\big|_\Gamma, t'' \models \varphi_M \quad .$$

We then show in the next section that, when L is regular, there exists a sound and complete detection procedure for every abstract behavior enjoying this property. Moreover, the time and space complexity of this detection procedure is linear in the size of the representation of L.

The following propositions show that the (m, n)-completeness property is realistic for abstract behaviors considered in practice.

We first prove, for particular abstract behaviors describing sequences of abstract actions with no constraints other than dataflow constraints, that we have the property of (m, n)-completeness.

Proposition 1. *Let Y be a set of variables of sort Data.*
Let $\alpha_1, \ldots, \alpha_m \in T_{\mathrm{Action}}(\mathcal{F}_\Gamma, Y)$. Then the abstract behavior $M := \exists Y. \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$ has the property of $(m, 0)$-completeness.

Proofs of propositions and theorems can be found in [25].

We now show that more complex abstract behaviors, forbidding specific abstract actions, have this property.

For a behavior pattern λ, let R_λ denote the restriction of the abstraction relation R to abstraction with respect to λ. We say that two behavior patterns λ and λ' are *independent* iff: $R_\lambda \circ R_{\lambda'} = R_{\lambda'} \circ R_\lambda$. Then we get the following result.

Proposition 2. *Let* $M := \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\ \mathbf{U}\, \lambda_3(\overline{x_3})$ *be an abstract behavior where* Y *and* Z *are two disjoint sets of variables of sort Data,* $\overline{x_1}, \overline{x_3} \in Y$, $\overline{x_2} \in Z$, *and where* $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ *and* λ_2 *is independent from* λ_3. *Then* M *has the property of* $(2,1)$*-completeness.*

In practice, as illustrated in Sect. 7, most signatures are disjunctions of formulas of the form: $\exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$, from Proposition 1, or of the form:

$$\exists Y. \lambda_1\,(\overline{x_1}) \wedge \neg\,(\exists Z_1.\, \lambda\,(\overline{z_1}))\ \mathbf{U}\, \lambda_2\,(\overline{x_2}) \wedge \neg\,(\exists Z_2.\, \lambda\,(\overline{z_2}))\ \mathbf{U}\ \ldots \lambda_k\,(\overline{x_k})$$

where λ is independent from $\lambda_2, \ldots, \lambda_k$. From the proof of Proposition 2, we conjecture that the last formula has the property of $(k,1)$-completeness.

The independence condition is not necessary in general, in order to guarantee that such abstract behaviors have a property of (m,n)-completeness for some m and n, but absence of this condition results in significantly higher values of m and n.

Fundamentally, by Definition 7, detection of an abstract behavior is decomposed into two independent steps: an abstraction step followed by a verification step. The first step computes the abstract forms of the program traces while the second step applies usual verification techniques in order to decide whether one of the computed traces verifies the FOLTL formula defining the abstract behavior. However, when using the (m,n)-completeness property to bypass the general intractability of the abstraction step, this relies on computing a set $\left\{t \in T_{\mathrm{Trace}}(\mathcal{F})\,,\, R^{\leq n}(t) \models \varphi_M\right\}$ and then intersecting it with $R^{\leq m}(L)$. So we lose the previous decomposition, thereby preventing us from leveraging powerful techniques from the model checking theory. We therefore show that, in the previous proposition, (m,n)-completeness allows us to nonetheless preserve that decomposition, so that the abstraction step now becomes decidable.

Theorem 2. *Let* M *be an abstract behavior defined by a formula* $\varphi_M = \exists Y.\ \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$ *where* Y *and* Z *are disjoint sets of variables of sort Data,* $\overline{x_1}, \overline{x_3} \in Y$, $\overline{x_2} \in Z$, *and where* $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ *and* λ_2 *is independent from* λ_3. *Then, for any set of traces* $L \subseteq T_{\mathrm{Trace}}(\mathcal{F}_\Sigma)$, L *exhibits* M *iff:*

$$\exists t \in\ R_{\lambda_2}\!\downarrow (R^{\leq 2}(L))\big|_\Gamma\,,\, t \models \varphi_M\ .$$

When both the abstraction relation R and the relation $R_{\lambda_2}\!\downarrow$ are rational, the set $R_{\lambda_2}\!\downarrow (R^{\leq 2}(L))$ is computable and regular, and detection then boils down to a classical model checking problem. In the general case, $R_{\lambda_2}\!\downarrow$ is not *rational*, but

in our experimentations, the behavior pattern λ_2 is defined by sets A_i and B_i where A_i contains traces made of a single action and $B_i = \{\epsilon\}$. Thus constructing a transducer realizing the relation $R_{\lambda_2}\downarrow$ is straightforward.

Remark 2. An equivalent definition of infection could consist in compiling the abstract behavior, that is computing the set $\pi_\Gamma^{-1}(M)\downarrow_{R^{-1}}$ of concrete traces exhibiting M. Then a set of traces L would exhibit M iff one of its traces is in this set. This definition seems more intuitive: rather than abstracting a trace and comparing it to an abstract behavior, we check whether this trace is an implementation of the behavior. However, this approach would require to first compute the compiled form of the abstract behavior, $\pi_\Gamma^{-1}(M)\downarrow_{R^{-1}}$, which is not generally computable and whose representation can quickly have a prohibitive complexity stemming from the interleaving of behavior patterns occurrences (especially when traces realizing the behavior patterns are complex) and from the variable instantiations.

6 Detection Complexity

The detection problem, like the more general problem of program analysis, requires computing a partial abstraction of the set of analyzed traces. In practice, in order to manipulate this set, we consider a regular approximation of it i.e., a tree automaton. Moreover, in practice, as seen in Sect. 4, the abstraction relation is rational, which entails the decidability of detection.

Theorem 3. *Let R be an abstraction relation, such that R and R^{-1} are rational. There exists a detection procedure deciding whether L exhibits M, for any regular set of traces $L \subseteq T_{\text{Trace}}(\mathcal{F}_\Sigma)$ and for any regular abstract behavior M having the property of (m,n)-completeness for some positive integers m and n.*

Definition 9. *Let M be an abstract behavior having the property of (m,n)-completeness. The* set of traces n-reliably realizing *M w.r.t an abstraction relation R is the set $\{t \in T_{\text{Trace}}(\mathcal{F}) \mid \forall t' \in R^{\leq n}(t)\big|_\Gamma, t' \models \varphi_M\}$.*

Using the set of traces n-reliably realizing M, when $T_{\text{Action}}(\mathcal{F})$ is finite, we get the following detection complexity, which is linear in the size of the automaton recognizing the program set of traces, a major improvement on the exponential complexity bound of [10].

Theorem 4. *Let R be an abstraction relation such that R and R^{-1} are rational. Let τ be a tree transducer realizing R. Let M be a regular abstract behavior with the property of (m,n)-completeness and A_M be a tree automaton recognizing the set of traces n-reliably realizing M w.r.t. R. Deciding whether a regular set of traces L, recognized by a tree automaton A, exhibits M takes $O\left(|\tau|^{m\cdot(m+1)/2} \times |A| \times |A_M|\right)$ time and space.*

7 Information Leak Behaviors

Abstraction can be applied to detection of generic threats, and in particular to detection of sensitive information leak. Such a leak can be decomposed into two steps: capturing sensitive information and sending this information to an exogenous location. The captured data can be keystrokes, passwords or data read from a sensitive network location, while the exogenous location can be the network, a removable device, etc. Thus, we define a behavior pattern $\lambda_{\text{steal}}(x)$, representing the capture of some sensitive data x, and a behavior pattern $\lambda_{\text{leak}}(x)$, representing the transmission of x to an exogenous location. Moreover, since the captured data must not be invalidated before being leaked, we define a behavior pattern $\lambda_{\text{inval}}(x)$, which represents such an invalidation.

Finally, the captured data is usually not leaked in its raw form, so we take into account transformations of this data via the behavior pattern $\lambda_{\text{depends}}(x, y)$ which denotes a dependency of x on y. For instance, x may be a string representation of y, or x may be an encryption or an encoding of y.

Then, in order to account for one such transformation of the stolen data, we define the information leak abstract behavior:

$$M := \exists x, y.\, \lambda_{\text{steal}}(x) \wedge \neg\lambda_{\text{inval}}(x)\ \mathbf{U}\ \lambda_{\text{depends}}(y, x) \wedge \mathbf{U}\ \lambda_{\text{leak}}(y)\ .$$

We consider the following definitions of the four behavior patterns involved, after looking at several malware samples, like keyloggers, sms message leaking applications or personal information stealing mobile applications:

– keystroke capture functionality:

$$\begin{gathered}\lambda_{\text{steal}}(x) := GetAsyncKeyState(x) \vee \\ (RegisterDev(\texttt{KBD}, \texttt{SINK}) \odot GetInputData(x, \texttt{INPUT})) \\ \vee(\exists y.\, SetWindowsHookEx(y, \texttt{WH\_KEYBOARD\_LL}) \wedge \\ \neg UnhookWindowsHookEx(y)\mathbf{U}\, HookCalled(y, x)) \\ \vee \exists y. TelephonyManager\_getDeviceId(x, y)\end{gathered}$$

– network send functionality:

$$\begin{gathered}\lambda_{\text{leak}}(x) := \exists y, z.\, sendto(z, x, y) \vee \exists y, z.\, (connect(z, y) \wedge \neg close(z) \\ \mathbf{U}\, send(z, x)) \vee \exists c, s.\, HttpURLConnection\_getOutputStream(s, c) \wedge \\ \neg OutputStream\_close(s)\mathbf{U}\, OutputStream\_write(s, x)\end{gathered}$$

– overwriting or freeing:

$$\lambda_{\text{inval}}(x) := free(x) \vee \exists y.\, sprintf_0(x, y) \vee GetInputData(x, \texttt{INPUT}) \vee \ldots$$

– dependences:

$$\begin{gathered}\lambda_{\text{depends}}(x, y) := sprintf_0(x, y) \vee \exists s.\, sprintf_1(x, s, y) \\ \vee \exists sb.\, StringBuilder\_append(sb, y) \odot SB\_toString(x, sb)\ .\end{gathered}$$

8 Experiments

Our goal is to detect the information leak behavior M defined in the previous section. In order to perform behavior pattern abstraction and behavior detection in the presence of data, we use the CADP toolbox [19], which allows us to manipulate and model-check communicating processes written in the LOTOS language. CADP features a verification tool, which allows on-the-fly model checking of formulas expressed in the MCL language, a fragment of the modal mu-calculus extended with data variables, whose FOLTL logic used in this paper is a subset.

We first represent the program set of traces as a CADP process, using a program control flow graph obtained by static analysis (see [21] and [23,24]). Regularity of the set of traces is enforced by limiting recursion and inlining function calls, an approximation that can be deemed safe with respect to the abstract behaviors to detect. Note that there are two shortcomings to regular approximation. First, approximation of conditional branches by nondeterministic branches may result in false positives, especially when the program code is obfuscated. And second, failure to identify data correlations during dataflow analysis can result in false negatives. However, this does not significantly impact our detection results.

Now, as expressed in Theorem 2, detection of the information leak abstract behavior M can be broken down into two steps: abstracting the set of traces L by computing $R_{\lambda_{\text{inval}}}\!\downarrow\big(R^{\leq 2}(L)\big)$ and then verifying whether an abstracted trace matches the abstract behavior formula.

So, we can simulate the abstraction step in CADP and delegate the verification step to the *evaluator4* module. For this, we represent the set of traces L of a given program by a system of communicating processes expressed in LOTOS, with a particular gate on which communications correspond to library calls. Then, computation of $R^{\leq 2}(L)$ is performed by synchronization with another LOTOS process which simulates the transducer realizing the abstraction. Moreover, the relation $R_{\lambda_{\text{inval}}}\!\downarrow$ is rational and can also be simulated by process synchronization in CADP.

For each malware sample we tested, we successfully ran *evaluator4* on the resulting process representing $R_{\lambda_{\text{inval}}}\!\downarrow(R^{\leq 2}(L))$, in order to detect the information leak abstract behavior defined in the previous section.

We essentially applied our approach to two case studies. The first one comes from a study on the detection rate of keylogger programs by existing antivirus [27], which shows a high failure rate. For an example of a typical keylogger for test, see [25]. From different keyloggers written in C for Windows, we constructed abstract behaviors of keylogger features. Then, tests we ran on keyloggers to know whether we are able to detect information leaking were successful.

Another example comes from an Android application for cell-phone named `SMS_Replicator_Secret`, which forwards received SMS to the attacker. This application defines a class `SMSReceiver` with a particular method `OnReceive (Context context, Intent intent)`. It then requests Android systems through its file metadata, to execute `OnReceive` on each SMS received or sent. We

extracted from this application abstract behaviors corresponding to SMS leaks. Unlike the previous case study, we ran partial tests because of the difficulty to set up an Android platform. They were successful.

9 Conclusion

We presented an original approach for detecting high-level behaviors in programs, describing combinations of functionalities and defined by first-order temporal logic formulas. Behavior patterns, expressing concrete realizations of functionalities, are also defined by first-order temporal logic formulas. Abstraction of these functionalities in program traces is performed by term rewriting. Validation of the abstracted traces with respect to some high-level behavior is performed via usual model checking techniques. In order to address the general intractability of the problem of constructing the normal form trace set for a given program, we have identified a property of practical high-level behaviors allowing us to avoid computing normal forms and yielding a linear time detection algorithm.

Abstraction is a key notion of our approach. Providing an abstracted form for program traces and behaviors allows us to be independent of the program implementation and to handle similar behaviors in a generic way, making this framework robust with respect to variants. The fact that high-level behaviors are combinations of elementary patterns enables us to efficiently summarize and compact the possible combinations likely to compose suspicious behaviors. Moreover, high-level behaviors and behavior patterns are easy to update since they are expressed in terms of basic blocks.

Our approach is at an early stage. We think that the theoretical results on behavioral analysis presented here are promising. Applicability of our detection technique could be further enhanced by automating construction of reference behavior patterns, for example using mining techniques as in [28].

Acknowledgements. We would like to thank Stephan Merz for fruitful discussions on temporal logics.

References

1. Cohen, F.: Computer viruses: Theory and experiments. Computers and Security 6(1), 22–35 (1987)
2. Le Charlier, B., Mounji, A., Swimmer, M.: Dynamic detection and classification of computer viruses using general behaviour patterns. In: International Virus Bulletin Conference, pp. 1–22 (1995)
3. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: IEEE Symposium on Security and Privacy, pp. 144–155. IEEE Computer Society (2001)
4. Morales, J., Clarke, P., Deng, Y., Kibria, G.: Characterization of virus replication. Journal in Computer Virology 4(3), 221–234 (2007)

5. Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. In: Symposium on Requirements Engineering for Information Security (2001)
6. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting Malicious Code by Model Checking. In: Julisch, K., Kruegel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
7. Singh, P.K., Lakhotia, A.: Static verification of worm and virus behavior in binary executables using model checking. In: Information Assurance Workshop, pp. 298–300. IEEE Computer Society (2003)
8. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A Layered Architecture for Detecting Malicious Behaviors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 78–97. Springer, Heidelberg (2008)
9. Bayer, U., Milani Comparetti, P., Hlauscheck, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: 16th Symposium on Network and Distributed System Security, NDSS (2009)
10. Jacob, G., Debar, H., Filiol, E.: Malware Behavioral Detection by Attribute-Automata Using Abstraction from Platform and Language. In: Balzarotti, D. (ed.) RAID 2009. LNCS, vol. 5758, pp. 81–100. Springer, Heidelberg (2009)
11. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. In: 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 377–388. ACM (2007)
12. Kinder, J., Zuleger, F., Veith, H.: An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 214–228. Springer, Heidelberg (2009)
13. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D.: BitScope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University (2007)
14. Security Issue on AMO, `http://blog.mozilla.com/addons/2010/02/04/please-read-security-issue-on-amo`
15. Aftermath of the Droid Dream Android Market Malware Attack, `http://nakedsecurity.sophos.com/2011/03/03/droid-dream-android-market-malware-attack-aftermath/`
16. Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: A sandbox for portable, untrusted x86 native code. In: 30th IEEE Symposium on Security and Privacy (S&P 2009), pp. 79–93. IEEE Computer Society (2009)
17. Kröger, F., Merz, S.: Temporal Logic and State Systems. Texts in Theoretical Computer Science. An EATCS Series. Springer (2008)
18. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
19. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
20. Chen, F., Roşu, G.: MOP: An Efficient and Generic Runtime Verification Framework. In: Object-Oriented Programming, Systems, Languages and Applications, pp. 569–588. ACM (2007)

21. Beaucamps, P., Gnaedig, I., Marion, J.-Y.: Behavior Abstraction in Malware Analysis. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 168–182. Springer, Heidelberg (2010)
22. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), http://www.grappa.univ-lille3.fr/tata
23. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: IEEE Symposium on Security and Privacy, pp. 32–46. IEEE Computer Society (2005)
24. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based Spyware Detection. In: Proceedings of the 15th USENIX Security Symposium (2006)
25. Beaucamps, P., Gnaedig, I., Marion, J.Y.: Behavior Analysis of Malware by Rewriting-based Abstraction - Extended Version. HAL-INRIA Open Archive Number inria-00594396 (2011)
26. Gilleron, R., Tison, S.: Regular Tree Languages and Rewrite Systems. Fundamenta Informaticae 24, 157–176 (1995)
27. Devine, C., Richaud, N.: A study of anti-virus' response to unknown threats. In: 18th Conference of the European Institute for Computer Anti-Virus Research, EICAR (2009)
28. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 5–14. ACM (2007)