Abstraction by Term Rewriting for Malware Behavior Analysis - Extended Version

Philippe Beaucamps, Isabelle Gnaedig, Jean-Yves Marion

Abstraction by Term Rewriting for Malware Behavior Analysis

Philippe Beaucamps, Isabelle Gnaedig, Jean-Yves Marion

INPL - INRIA Nancy Grand Est - Nancy-Université - LORIA
Campus Scientifique - BP 239 F54506 Vandoeuvre-lès-Nancy Cedex, France

Email: {Philippe.Beaucamps, Isabelle.Gnaedig, Jean-Yves.Marion}@loria.fr

Abstract. We propose a formal approach for behavioral analysis of programs based on dynamic analysis. It works by abstracting given behavior patterns in a high level trace language of a program and then, by comparing the abstract trace forms to a signature expressing reference abstract malicious behaviors. Abstraction is performed by term rewriting using rules on terms with variables, which allows to take into account parameters on functionalities expressed in behaviors. This technique allows to handle interleaved behaviors. Successfully applied to malware detection, it allows in particular to analyze some kind of data leak.

1 Introduction

Several approaches may be combined to analyze the behavior of a program. In a runtime approach, a program is executed without control and a trace of the execution is built by capturing only data of interest. As a result, part of the program behavior remains unknown. In a static approach, the program code is analyzed without executing it: using this approach, a more exhaustive representation of the program can be constructed. However, when we try to analyze a binary code, which is nowadays pratically always the case in malware detection, the program code is not available. So it is not possible to apply usual static analysis in a direct way. Indeed, determining the program code is an intractable problem, for two main reasons. First, there are indirect jumps, like the instruction `jmp eax` which jumps to the value pointed by the x86-register `eax`. Second, a program may use complex code protection, for example by changing dynamically its code. In a dynamic approach, the program is executed and its execution is controlled, allowing to bypass some shortcomings of static analysis and to expand code coverage. Thus, in order to compute a representation of the program behavior, defined as its set of execution traces, one may consider combining both static and dynamic approaches.

Behavior analysis was introduced by Cohen's seminal work [3] in order to detect malware and in particular unknown malware. In general, a behavior is described by a sequence of system calls and recognition is based on finite state automata over words, see for example [9,12,11]. This approach is quite limited, which motivated some recent works. In [7], the authors use attribute automata, at

the price of an exponential time complexity detection procedure. Model-checking is used in [2,8,13] to track data. But none of these works consider functional polymorphism. Moreover they do not tackle either the problem of constructing a high-level view of a program, which limits their applicability. In [10], functional polymorphism is considered by preprocessing execution traces and transforming them into a high-level representation which captures their semantic meaning. But as these approaches deal with the execution trace being observed, they analyze a single behavior at a time.

In this paper, we propose an approach allowing program analysis by abstracting behavior components in program traces. Starting from a given set of execution traces, we compute an abstract representation of the program behavior, which is a representation of the set of abstract forms of its execution traces. This abstraction amounts to identifying a set of known functionalities which we describe by behavior patterns, and to rewriting these behavior patterns into abstract functionality symbols. Thus, the abstract form of an execution trace is defined in terms of these functionalities and not anymore in terms of observed actions, which are low-level and therefore less reliable. This abstraction can be used in two scenarios:

- Detection of malicious behaviors: the signature of a malicious behavior is expressed in terms of abstract functionalities, making it implementation independent and appropriate to detect future variants of the same behavior. Given some program, we then assess whether one its execution traces exhibits a sequence of known functionalities, in a way specific to one of the predefined malicious behaviors.
- Analysis of suspicious programs: abstraction provides a simple and high-level representation of a program behavior which is more suitable for manual analysis, signature generation, analysis of behavioral similarity with other malware, etc.

Motivation and Roadmap We want to recognize particular malicious behaviors and to be resilient to mutations in the way these behaviors are realized. Such mutations may come from variant creation or simply from obfuscation techniques, packing techniques, etc.

Therefore, we define a set of functionalities that compose the behaviors to recognize. We express a functionality in terms of elementary actions. A functionality is defined by a logic formula describing how the functionality is realized. For example, we may define the functionality of capturing keystrokes by the formula:

$$\varphi_{kb\,capture} := \exists x.\, RegisterRawInputDevices\,(\texttt{GENERIC\_KBD}, \texttt{SINK}) \wedge \top\, \mathbf{U}\, GetRawInputData\,(x, \texttt{INPUT})$$

where `GENERIC_KBD`, `SINK` and `INPUT` are particular constants that guarantee that keyboard events are indeed captured. Similarly, we may define the functionality of writing to a file by the formula: $\varphi_{write\,file} := \exists x, y.\, fwrite\,(x, y)$.

Moreover, a functionality, and by extension a behavior, may be realized in several ways. In our example, the functionality of capturing keystrokes may also be realized by the formula $\exists x.\, GetAsyncKeyState\,(x)$.

Now, when monitoring a program, an execution trace is captured, representing a sequence of actions. For instance, we may observe the following trace:

$$t = RegisterRawInputDevices\,(\texttt{GENERIC\_KBD}, \texttt{SINK}) \cdot fopen\,(1,2) \cdot GetRawInputData\,(3, \texttt{INPUT}) \cdot fwrite\,(1,3)\,.$$

A subtrace of t validates the formula $\varphi_{kb\,capture}$ and therefore realizes the functionality of capturing keystrokes. Similarly, a subtrace of t validates the formula $\varphi_{write\,file}$ and therefore realizes the functionality of writing to a file. The combination of both functionalities precisely makes this trace suspect. Thus, we are able to detect a malicious behavior by identifying in a trace occurrences of the functionalities composing this behavior. Detection of a behavior is performed at the functionality level, that is at a higher level than the level of elementary actions.

On another hand, we want to be generic with respect to the way functionalities are realized. To that end, we represent functionalities by abstract symbols in a set Γ. We define a behavior on Γ. We then need to define an abstraction relation R_α allowing to transform a raw trace into an abstract trace on Γ. This relation is defined with the help of a rewriting system whose left handsides of rules precisely identify occurrences of functionalities. A trace t then exhibits the behavior M if:

$$\exists u \in M,\, R_\alpha\,(t) = t' \cdot u \cdot t''.$$

For instance, using the previous functionalities, we associate the functionality of capturing keystrokes to a unary function symbol $\lambda_{kb\,capture}$ in Γ and the functionality of writing to a file to a binary function symbol $\lambda_{write\,file}$ in Γ. So here, $\lambda_{kb\,capture}$ denotes several ways of performing a keystroke capture. We then define the information leak behavior by the set of terms:

$$\{t \,|\exists x, y, t',\, t = \lambda_{kb\,capture}\,(x) \cdot t' \cdot \lambda_{write\,file}\,(y,x)\}\,.$$

Moreover, we add parameters to abstraction symbols to describe the data they manipulate. Finally, rather than working on single traces, as is the usual case in the domain of malware detection, we consider unbounded sets of traces, that may come, as said previously, from a runtime analysis but also from a static or dynamic analysis. This allows constructing an abstract representation of a program behavior and we can detect in linear time if this program exhibits a given behavior. Furthermore, as was just illustrated, this formalism is particularly adapted to the protection against generic threats like the leak of sensitive information.

Previous works In [1], we already proposed to abstract program trace languages with respect to behavior patterns, in order to detect known reference behaviors or to better understand the programs. Patterns were defined by closed string

rewriting systems which did not allow the described actions to have parameters. Moreover abstraction rules replaced identified patterns by abstraction symbols in the original trace, precluding a further detection of patterns interleaved with rewritten one. For instance, a behavior pattern identifying a sequence ab was associated to the string rewriting system rewriting any string $a \cdot u \cdot b$ into λ, which corresponds to the following term rewriting system R:

$$\begin{gathered} a \to q \\ q \cdot z \cdot x \to q \cdot x \\ q \cdot b \cdot x \to \lambda \cdot x \end{gathered}$$

If we define a new behavior pattern by the sequence cd and associate it to a symbol λ', then the trace $acdb$ is abstracted either into λ, or in λ', but never in $\lambda\lambda'$ or $\lambda'\lambda$. Therefore, the goal of this paper is to propose a new formalism which would allow:

- to account for interleaved behavior patterns, in order to rewrite for instance $a \cdot c \cdot d \cdot b$ into $\lambda \cdot \lambda'$;
- to express data constraints on action parameters, for instance by requiring that actions c and d use the same objet;
- to give parameters to behavior patterns themselves in order to then analyze the dataflow in abstracted traces.

Another main difference with the previous result [1] is that we may be able to detect information leaks, i.e. to prevent unauthorized disclosure or modifications of information. Indeed, we are able to dynamically track some data flow, as we shall see in a short while. Here again, our point of view is that dynamic analysis may be complementary to static analysis and formal methods. Verification of systems is now available for critical modules using test methods, model-checking, or formal proofs. This is necessary to obtain trusted systems with fine grained compartmentalization. Nevertheless, and despite formal verification, we cannot produce flawless software. So it is still necessary to enforce security policy at runtime. Of course, dynamic analysis at runtime is costly - see for example SeLinux [6] which implements mandatory access control on Linux - but we should pay the price of security as suggested by Tanenbaum [14].

2 Background

Term Algebras Let $S = \{TRACE, ACTION, DATA\}$ be a set of sorts and $\mathcal{F} = \mathcal{F}_t \cup \mathcal{F}_a \cup \mathcal{F}_d$ be an S-sorted signature, where $\mathcal{F}_t$, $\mathcal{F}_a$, $\mathcal{F}_d$ are mutually distincts and:

- $\mathcal{F}_t = \{\epsilon, \cdot\}$, with: $\epsilon : TRACE$, $\cdot : ACTION \times TRACE \to TRACE$.
- $\mathcal{F}_a$, a finite set of function symbols or constants, with signature $DATA^n \to ACTION$, $n \in \mathbb{N}$, which describe actions.
- $\mathcal{F}_d$, a finite set of constants of type $DATA$, which describe data.

We distinguish the sort $ACTION$ from the sort $TRACE$. However, 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 denote by $T(\mathcal{F}, X)$ the set of S-sorted terms over a set X of S-sorted variables. For any sort $s \in S$, we denote by $T_s(\mathcal{F}, X)$ the restriction of $T(\mathcal{F}, X)$ to terms of sort s and we denote by X_s the subset of variables of X of sort s.

If $f \in \mathcal{F}$ is a symbol of arity $n \in \mathbb{N}$, we denote by $f(\overline{x})$ a term $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are variables.

Substitutions are defined as usual (see appendix). 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 closed substitutions over X is denoted by $Subst_X$.

We partition $\mathcal{F}_a$ in a set Σ of symbols denoting concrete actions and a set Γ of symbols denoting abstract actions identifying functionalities to be abstracted. When considering purely concrete (resp. abstract) terms, we use the notation $\mathcal{F}_\Sigma = \mathcal{F} \setminus \Gamma$ (resp. $\mathcal{F}_\Gamma = \mathcal{F} \setminus \Sigma$).

We define in a natural way the projection and the concatenation of traces or sets of traces, with the notation $t \cdot t'$ for the concatenation of traces t and t', and the notations $t|_{\Sigma'}$ or $\pi_{\Sigma'}(t)$ for the projection of a trace t on an alphabet $\Sigma' \subseteq \Sigma \cup \Gamma$ (see appendix).

Program Behavior The abstract representation of the program we want to observe is chosen to be a set of traces of this program. When executing a program, the captured data is represented by the alphabets Σ and $\mathcal{F}_d$. In this paper, we consider more specifically the case where library calls are captured along with their arguments. Σ represents the finite set of library calls and constants from $\mathcal{F}_d$ 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 $T_{TRACE}(\mathcal{F}_\Sigma)$. Similarly, a program behavior is defined by the set of its execution traces, that is a possibly infinite set of traces of $T_{TRACE}(\mathcal{F}_\Sigma)$. For instance, the term $fopen(1,2) \cdot fwrite(1,3)$ represents the execution trace of the file open call $fopen(1,2)$ followed by the file write call $fwrite(1,3)$, where $1 \in \mathcal{F}_d$ identifies the file handle returned by the first call, $2 \in \mathcal{F}_d$ identifies the file path and $3 \in \mathcal{F}_d$ identifies the written data.

First-Order LTL (FOLTL) Temporal Logic We consider an adaptation of the LTL temporal logic (see Appendix A.2), where atomic predicates are terms and may have variables. This corresponds to the First-Order Linear Temporal Logic defined in [NEED REF Stefan Merz]. More precisely, let X be a set of variables of sort $DATA$ and $AP = T_{ACTION}(\mathcal{F}_\Sigma, X)$ be the set of atomic propositions.

An FOLTL formula is defined as follows:

- 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.

Notation $\varphi_1 \odot \varphi_2$ will stand for $\varphi_1 \wedge \mathbf{X}\varphi_2$.

We say that an 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 closed substitution over Y. We naturally define the application of σ to an FOLTL formula φ by the formula $\varphi\sigma$ where any free variable x in φ which is in Y has been replaced by its value $\sigma(x)$.

As with LTL, a formula is validated on infinite sequences of sets of atomic predicates, denoted by $\xi = (a_0, a_1, \ldots) \in \left(2^{AP}\right)^{\omega}$. $\xi \models \varphi$ (ξ validates φ) is defined in the same way as for the LTL logic, with the following additional rules:

– $\xi \models \exists Y.\varphi$ iff there exists a substitution $\sigma \in Subst_Y$ such that $\xi \models \varphi\sigma$;
– $\xi \models \forall Y.\varphi$ iff, for any substitution $\sigma \in Subst_Y$, $\xi \models \varphi\sigma$.

In our context, a formula is validated over traces of $T_{TRACE}(\mathcal{F}_\Sigma)$ identified with sequences of singleton sets of atomic predicates. A trace $t = a_0 \cdots a_n$ is identified with the infinite sequence of sets of atomic predicates $(\{a_0\}, \ldots, \{a_n\}, \{\}, \{\}, \ldots)$, and t validates φ, which is denoted by $t \models \varphi$, iff $(\{a_0\}, \ldots, \{a_n\}, \{\}, \{\}, \ldots) \models \varphi$.

For examples, see the appendix.

Tree Automata and Tree Transducers Tree automata and tree transducers are defined as usual (see Appendix and [4]). A tree language is regular if recognized by some tree automaton and a binary relation is rational if realized by some tree transducer.

3 Trace Abstraction

The problem under study can be formalized in the following way. First, we need to define a set $\{B_i\}$ of behavior patterns, where each behavior pattern represents a (possibly infinite) set of terms from $T_{TRACE}(\mathcal{F}_\Sigma)$. Second, we need to define an abstraction relation R_α allowing to schematize a trace by recognizing behavior patterns from $\{B_i\}$ in that trace. Finally, given some program p coming with either a finite set L of traces (runtime analysis or simulation scenarios) or an infinite set L of traces (static analysis scenario, for instance by using the control flow graph), we examine the following problems:

– Detection problem: we define M as an abstract malicious behavior representing a set of terms from $T_{TRACE}(\mathcal{F}_\Gamma)$. We then consider that p is infected by M iff $R_\alpha(L) \cap M \neq \emptyset$. Our goal is to find an effective and efficient method deciding whether p is infected by M or not.
– Analysis problem: we compute a representation of $R_\alpha(L)$ which is an abstract, simple and high-level description of the program behavior, which may therefore be used in manual analysis, behavioral similarity analysis, etc.

3.1 Behavior Patterns

A behavior pattern describes a functionality we want to recognize in a program trace, for instance: writing to system file, sending a mail or pinging a remote host. These functionalities can be realized in different ways: by using different system calls, different library calls, different programming languages, etc. A behavior pattern may therefore be defined as the set of traces realizing the functionality it describes.

Example 1. Let's consider the behavior pattern describing a ping. One of its realizations consists in calling the *socket* function with the particular parameter `IPPROTO_ICMP` and then to call the *sendto* function with the particular parameter `ICMP_ECHOREQ` describing data to be sent. Between these two calls, the socket should not have been freed or reallocated.

Assume that constants α and β in $\mathcal{F}_d$ identify parameters `IPPROTO_ICMP` and `ICMP_ECHOREQ`, that the first parameter of *socket* is the created socket and the second parameter is the network protocol, that the first parameter of *sendto* is the used socket, the second parameter is the sent data and the third parameter is the target, and the unique parameter of *closesocket* is the freed socket.

This realization of the ping is therefore intuitively described by the FOLTL formula: $\exists x, y.\, socket\,(x, \alpha) \wedge (\neg closesocket\,(x)\ \mathbf{U}\ sendto\,(x, \beta, y))$.

In this realization, a position is distinguished from which the functionality has been performed, that is just after *sendto*. This position will be chosen as the insertion position for the abstraction symbol. Indeed, this position is the most logical one to stick to the trace semantics. Furthermore, when behavior patterns appear interleaved, this position allows to define the order in which their functionalities were realized (see appendix).

Moreover, we may want to add a parameter to the abstraction symbol, typically to express dataflow constraints in a signature. For instance, the abstraction symbol could take a unique parameter which denotes the ping target: then a signature for a denial of service could be defined as a sequence of 100 pings with the same target.

Definition 1 (Behavior Pattern). *A behavior pattern is a set of traces* $B \subseteq T_{TRACE}\,(\mathcal{F}_\Sigma)$ *validating a closed FOLTL formula* φ *:*

$$B = \{t \in T_{TRACE}\,(\mathcal{F}_\Sigma) \mid t \models \varphi\}\,.$$

Example 2. The behavior pattern describing a ping is described by the formula:

$$\exists x, y.\, socket\,(x, \alpha) \odot \neg closesocket\,(x)\,\mathbf{U} sendto\,(x, \beta, y)$$

which is satisfied by traces in the following set:

$$\bigcup_{\sigma \in Subst_X} (socket\,(x, \alpha)\,\sigma \cdot \\ T_{TRACE}\,(\mathcal{F}_\Sigma) \setminus (T_{TRACE}\,(\mathcal{F}_\Sigma) \cdot closesocket\,(x)\,\sigma \cdot T_{TRACE}\,(\mathcal{F}_\Sigma)) \cdot \\ sendto\,(x, \beta, y)\,\sigma)$$

3.2 Trace Abstraction

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, that we call abstraction symbol, at the position after which the behavior pattern functionality has been performed. This symbol can have parameters describing those used by the behavior pattern occurrence. As said in the introduction, replacing behavior patterns occurrences by their abstraction symbol does not allow handling interleaving of behavior patterns occurrences, hence our choice of inserting the abstraction symbol within the occurrence.

Example 3. Back to the ping behavior pattern, we associate it with a unary abstraction symbol λ_{ping} whose only parameter describes the ping target.

Moreover, we consider a second realization of the ping that uses the function $IcmpSendEcho : DATA \rightarrow ACTION$ whose argument represents the ping target. The ping behavior pattern is then described by the formula:

$$\begin{gathered}(\exists x, y.\, socket\,(x, \alpha) \odot \neg closesocket\,(x)\, \mathbf{U} sendto\,(x, \beta, y)) \\ \vee\, (\exists x.\, IcmpSendEcho\,(x))\,.\end{gathered}$$

For the first realization, the *sendto* action effectively performs the ping, so we wish to rewrite the trace $socket\,(1, \alpha) \cdot gethostbyname\,(2) \cdot sendto\,(1, \beta, 3) \cdot closesocket\,(1)$ into: $socket\,(1, \alpha) \cdot gethostbyname\,(2) \cdot sendto\,(1, \beta, 3) \cdot \lambda_{ping}\,(3) \cdot closesocket\,(1)$.

Abstraction of the ping for this realization therefore corresponds to rewriting a trace 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:

$$\begin{gathered}A_1\,(x, y) = socket\,(x, \alpha) \cdot \\ (T_{TRACE}\,(\mathcal{F}_\Sigma) \setminus (T_{TRACE}\,(\mathcal{F}_\Sigma) \cdot closesocket\,(x) \cdot T_{TRACE}\,(\mathcal{F}_\Sigma))) \cdot \\ sendto\,(x, \beta, y)\,.\end{gathered}$$

$$B_1\,(x, y) = \{\epsilon\}\,.$$

For the second realization, we want to insert the action $\lambda_{ping}\,(x)$ after the $IcmpSendEcho\,(x)$ action. Abstraction of the ping for this realization therefore corresponds to rewriting a trace using the rule: $A_2\,(x) \cdot B_2\,(x) \rightarrow A_2\,(x) \cdot \lambda\,(y) \cdot B_2\,(x)$ where $A_2\,(x) = \{IcmpSendEcho\,(x)\}$ and $B_2\,(x) = \{\epsilon\}$.

The abstraction relation is therefore defined by decomposing the behavior pattern into a finite union of combinations 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 inside a trace should not impact the abstraction of an occurrence of the behavior pattern.

Definition 2 (Abstraction System). *Let $\lambda \in \Gamma$ be an abstraction symbol of arity $m \in \mathbb{N}$, X be a set of variables of sort $DATA$, $\overline{x}$ be a sequence of variables in X. An abstraction system over $T_{TRACE}\,(\mathcal{F})$ is a finite set of rewriting rules of the form:*

$$A_i\,(X) \cdot B_i\,(X) \rightarrow A_i\,(X) \cdot \lambda\,(\overline{x}) \cdot B_i\,(X)$$

where the sets $A_i\,(X)$ and $B_i\,(X)$ are sets of concrete traces of $T_{TRACE}\,(\mathcal{F}_\Sigma, X)$.

The system of rewriting rules that we use generates a reduction relation over $T_{TRACE}(\mathcal{F})$ such that filtering occurs on traces projected on Σ and the abstraction symbol is inserted after the last concrete action of a term in $A_i(X)$:

Definition 3. *The reduction relation over $T_{TRACE}(\mathcal{F})$ generated by a system with n rewriting rules $A_i(X) \cdot B_i(X) \rightarrow A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ is the rewriting relation $\rightarrow_{\mathcal{R}}$ such that, for all $t, t' \in T_{TRACE}(\mathcal{F})$:*

$$\begin{gathered} t \rightarrow_{\mathcal{R}} t' \\ \Leftrightarrow \\ \exists \sigma \in Subst_X, \exists u \in T_{TRACE}(\mathcal{F}), \exists p \in Pos(t), \\ \exists i \in [1..n], \exists a, b \in T_{TRACE}(\mathcal{F}), \\ a \in T_{TRACE}(\mathcal{F}) \cdot T_{ACTION}(\mathcal{F}_\Sigma), a|_\Sigma \in A_i(X)\sigma, b|_\Sigma \in B_i(X)\sigma, \\ t|_p = a \cdot b \cdot u \text{ and } t' = t[a \cdot \lambda(\overline{x})\sigma \cdot b \cdot u]_p. \end{gathered}$$

An abstraction relation with respect to a given behavior pattern is thus the reduction relation of such an abstraction system, where left members of the rules describe the set of occurrences of the behavior pattern.

Definition 4 (Abstraction). *Let B be a behavior pattern associated with an abstraction symbol $\lambda \in \Gamma$, of arity $m \in \mathbb{N}$. Let X be a set of variables of sort $DATA$. An abstraction relation with respect to this behavior pattern is the reduction relation over $T_{TRACE}(\mathcal{F})$ generated by an abstraction system composed of n rules $A_i(X) \cdot B_i(X) \rightarrow A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ which verify:*

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

The condition on the abstraction system guarantees that the set of instances of left members of the rewrites rules actually covers the set of occurrences of the behavior pattern.

Remark 1. The abstraction relation associated with a behavior pattern is not unique. We could chose to insert the abstraction symbol at any position. As was previously said, we chose to insert it at the position just after which the functionality has been effectively performed.

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

Definition 5. *Let C be a finite set of behavior patterns. An abstraction relation with respect to C is the union of abstraction relations with respect to each behavior pattern of C.*

From now on, if B is a behavior pattern defined using a closed FOLTL formula φ and associated to an abstraction symbol λ, we may describe it by the simplified notation $\lambda := \varphi$.

3.3 Total Abstraction

From Definition 4, an abstraction relation step with respect to a behavior pattern represents the abstraction of a single occurrence of the pattern. In the general case, and in particular in the case of infinite sets of traces, we want to analyze the set of completely abstracted traces. In other words, if R is an abstraction relation with respect to our set of behavior patterns, we want to define a total abstraction relation which computes the set $R{\downarrow}$ of normal forms of any trace with respect to R.

In the case of a finite set of traces L, abstraction does not terminate in general, since the same occurrence of a pattern can be abstracted an unbounded number of times. We therefore need to require that the same abstract action is not inserted twice after the same concrete action. In other words, if a term $t = t_1 \cdot t_2$ is abstracted into a term $t' = t_1 \cdot \alpha \cdot t_2$, where α is the inserted abstract action, and if t_2 starts with a sequence of abstract actions $\alpha_1 \cdots \alpha_n$, then the α_i are distinct from α: $\forall i \in [1..n]\,,\ \alpha_i \neq \alpha$.

Definition 6 (Terminating Abstraction). *The terminating abstraction relation for an abstraction relation $\mathcal{R}$ is the relation $\mathcal{R}'$ defined by:*

$$\begin{gathered}\forall t_1, t_2 \in T_{TRACE}\left(\mathcal{F}\right), \forall \alpha \in T_{ACTION}\left(\mathcal{F}_\Gamma\right), \\ t_1 \cdot t_2 \ \rightarrow_{\mathcal{R}'}\ t_1 \cdot \alpha \cdot t_2 \\ \Leftrightarrow \\ t_1 \cdot t_2 \ \rightarrow_{\mathcal{R}}\ t_1 \cdot \alpha \cdot t_2 \\ \textit{and} \quad \nexists (u, u') \in T_{TRACE}\left(\mathcal{F}_\Gamma\right) \times T_{TRACE}\left(\mathcal{F}\right),\ t_2 = u \cdot \alpha \cdot u'.\end{gathered}$$

In the case of an infinite set of traces L, the computation of $L{\downarrow_R}$ often relies on the computation of the set of descendants $R^*\left(L\right)$. However $R^*\left(L\right)$ is not computable in general [5], and the term rewriting system implementing the abstraction relation does not belong to any known class of term rewriting systems whose transitive closure preserves regularity (see [4]).

4 Detection Problem

We assume R is a terminating abstraction relation (see Definition 6). A malicious behavior is expressed in a purely abstract way, expressing the fact that its malicious nature does not come from its implementation but only from the sequence of functionalities that are performed. We therefore define a malicious behavior as the set of abstract traces that realize it.

Definition 7. *A malicious behavior is a set of terms of $T_{TRACE}\left(\mathcal{F}_\Gamma\right)$.*

Then, given some (possibly infinite) set of traces, the detection problem consists in deciding whether one of these traces exhibits a malicious behavior M.

Definition 8. *A set of traces L exhibits a malicious behavior M, denoted by $L \models M$, iff:*

$$L{\downarrow_R}|_\Gamma \cap \left(T_{TRACE}\left(\mathcal{F}_\Gamma\right) \cdot M \cdot T_{TRACE}\left(\mathcal{F}_\Gamma\right)\right) \neq \emptyset.$$

In other words, the set of abstract traces with respect to R contains a subtrace with a behavior of M. This definition relies on the construction of the set $L{\downarrow_R}$ of normal forms of traces from L, which is undecidable in the general case (see Section 3.3). But in the case of detection, computing the normal form seems unnecessary, as a partial abstraction of the set of traces should be enough to evaluate whether the program is malicious.

We therefore propose a detection algorithm relying on an under-approximation of the set of abstract traces, which, however, must be chosen carefully. For instance, it cannot consist in computing $R^{\leq n}(L)$, the set of descendants of L until the order n, for some n, as is shown by the following example.

Example 4. Let $\lambda_1 := a \wedge (\top \,\mathcal{U}\, d)$, $\lambda_2 := b$, $\lambda_3 := c$ be three behavior patterns associated to abstraction relations inserting the abstraction symbol after a, b and c respectively. Let $M = \lambda_1 \wedge (\neg\lambda_2 \,\mathcal{U}\, \lambda_3)$ be a malicious behavior. Assume there exists a bound n such that $L{\downarrow_R}$ may be approximated by $R^{\leq n}(L)$ in Definition 8 of the infection. The trace $t = a^{n-1} \cdot b \cdot c \cdot d$ is an example of a sane trace. 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 trace t is malicious.

The previous example shows that the set of partially abstracted traces $R^{\leq n}(L)$ is not sufficient to evaluate the malicious nature of the program as it contains contradictory traces that compromise detection, that is traces seemingly exhibiting a malicious behavior though a few additional abstraction steps would make them leave the signature. Consequently, we want to identify unreliably infected traces in $R^{\leq n}(L)$, while not having to reach normal forms. In fact, we identify a fundamental property we call (n, m)-completeness, verified by malicious behaviors in practice in the field of behavior detection. This property specifies that, to show that a program is infected, a necessary and sufficient condition is that there exists a partially abstracted trace, abstracted at most n times, that is infected and whose descendants up to the order m are still infected. We present a sound and complete detection procedure for every malicious behavior enjoying this property. Besides, when the set of traces L is regular, the complexity in time and space of this detection procedure is linear in the size of the representation of L (see Section 5).

Definition 9. *Let M be a malicious behavior. A partially abstracted trace $t \in T_{TRACE}(\mathcal{F})$ is* reliably infected *by M, iff: $\forall t' \in R^*(t)|_\Gamma, t' \in T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$.*

Deciding whether a (partially abstracted) trace is reliably infected is undecidable since $R^*(t)$ is undecidable. However, for some malicious behaviors, it is sufficient to observe the set of descendants until an order m, instead of $R^*(t)$.

From this observation, we suggest an equivalent expression of infection where L is infected by M if from a certain level of abstraction, there is a trace $t' \in R^{\leq n}(L)$ such that all descendants of t' with respect to R are malicious.

Definition 10 ((n, m)-completeness). *Let M be a malicious behavior and n and m be positive numbers. M has the property of* (n, m)-completeness *iff for*

any set of traces $L \subseteq T_{TRACE}(\mathcal{F}_\Sigma)$*:*

$$L \models M \\ \Leftrightarrow \\ \exists t' \in R^{\leq n}(L),\ R^{\leq m}(t')\big|_\Gamma \subseteq T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma).$$

When a malicious behavior M has the property of (n, m)-completeness, we can define the set of traces reliably infected by M in the following way.

Definition 11. *Let* M *be a regular malicious behavior having the property of* (n, m)*-completeness. The set of traces reliably infected by* M *with respect to an abstraction relation* R *is the set*

$$\left\{t' \mid R^{\leq m}(t')\big|_\Gamma \subseteq T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)\right\}.$$

The following theorems show that this property is realistic, that is malicious behaviors considered in practice indeed have a property of (n, m)-completeness.

We first prove that simple malicious behaviors describing sequences of abstract actions with no constraints other than dataflow constraints have the property of (n, m)-completeness. Examples of such malicious behaviors include $\lambda_1 \odot \lambda_2$ and $\exists x, y.\, \lambda_1(x) \odot \lambda_2(x, y) \odot \lambda_3$.

Theorem 1. *Let* X *be a set of variables of sort* $DATA$*. Let* $\alpha_1, \ldots, \alpha_n \in T_{ACTION}(\mathcal{F}_\Gamma, X)$*. Then the malicious behavior* $M := \exists X.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_n$ *has the property of* $(n, 0)$*-completeness.*

Proofs of the theorems are given in the extended version of this paper.

We now show that more complex malicious behaviors, expressing constraints on the abstract actions which appear interleaved with the malicious behavior have this property. Examples of such malicious behaviors include:

- $\exists x.\, \lambda_1(x) \wedge \neg\lambda_2(x)\ \mathbf{U}\ \lambda_3$: action $\lambda_2(x)$ is forbidden between actions $\lambda_1(x)$ and λ_3;
- $\exists x \forall y.\, \lambda_1(x) \wedge \neg\lambda_2(x, y)\ \mathbf{U}\ \lambda_3$: for some fixed x, we need to observe actions $\lambda_1(x)$ and λ_3 and no instance of action $\lambda_2(x, y)$ shall be observed between these actions.

We thus generalize by distinguishing two sets of variables: existential variables represented by the set X and universal variables represented by a set Y and that can only be used with negations.

Theorem 2. *Let* X*,* Y *be two disjoint sets of variables of sort* $DATA$*. Let* $\lambda_1, \lambda_2, \lambda_3 \in \mathcal{F}_\Gamma$ *be three abstraction symbols, with* $\lambda_2 \neq \lambda_1$ *and* $\lambda_2 \neq \lambda_3$*. Let* $\alpha_1 = \lambda_1(\overline{x_1})$*,* $\alpha_2 = \lambda_2(\overline{x_2})$*,* $\alpha_3 = \lambda_3(\overline{x_3})$ *be abstract actions such that* X *is exactly the set of variables appearing in* $\overline{x_1}$ *and* $\overline{x_3}$ *and variables of* $\overline{x_2}$ *are variables from* $X \cup Y$*. Then the malicious behavior* $M := \exists X \forall Y.\, \alpha_1 \wedge (\neg\alpha_2\ \mathbf{U}\ \alpha_3)$ *has the property of* $(3, 2)$*-completeness.*

These results can be generalized to more general forms of signatures, although we do not prove it here, as the proof is technical and similar to proofs of the previous theorems. For instance, we expect a signature $\exists x.\, \lambda_1(x) \odot \lambda_2 \wedge \neg\lambda_3(x)\ \mathbf{U}\ \lambda_4$ to have the property of $(4,1)$-completeness, or a signature $\exists x.\, \lambda_1(x) \wedge \neg\lambda_3(x)\ \mathbf{U}\ \lambda_2 \wedge \neg\lambda_3(x)\ \mathbf{U}\ \lambda_4$ to have the property of $(5,2)$-completeness.

Remark 2. An equivalent definition of infection could consist in compiling the malicious behavior, that is computing the set $\pi_\Gamma^{-1}(M)\downarrow_{R^{-1}}$ of concrete traces exhibiting this behavior. Then a set of traces L would exhibit this behavior if one of its subtraces is in this set. This definition seems more intuitive: rather than abstracting a trace and comparing it to an abstract malicious behavior, we check whether this trace is an implementation of the malicious behavior. However, this approach would require to first compute the compiled form of the malicious 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 realizations of the behavior patterns are complex) and from the instanciation of variables.

5 Rational Abstraction

The detection problem, like the more general problem of program analysis, requires to compute a partial abstraction of the set of analyzed traces. In practice, in order to manipulate such a set of traces, we consider a regular approximation of this set, i.e. we represent it by a tree automaton. Then, when it comes to effectively abstracting this set, i.e. constructing a representation of its partially abstract form, the formalism of tree transducers is a suitable approach with interesting formal (closure of union, intersection, complementation, composition, preservation of regularity) and computational properties. In practice, indeed, 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 to ensure that the abstraction relation is realizable by a tree transducer, in other words that it is a rational tree transduction.

Theorem 3. *Let B be a behavior pattern and R be an abstraction relation with respect to B defined from 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 realized by two transducers of size $O(|A_R|)$.*

Theorem 4. *Let R be an abstraction relation, such that R and R^{-1} are rational. There exists a detection procedure deciding if $L \models M$, for any regular set of traces L and for any regular malicious behavior M having the property of (n,m)-completeness for some positive integers n and m.*

With the set of traces reliably infected by M defined in Definition 11, we get the following detection complexity.

Theorem 5. *Let R be an abstraction relation such that R and R^{-1} are rational. Let τ be a bottom-up tree transducer realizing R. Let M be a regular malicious behavior with the property of (n, m)-completeness and A_M be a tree automaton recognizing the set of traces reliably infected by M with respect to R.*

Deciding if a regular set of traces L, recognized by a tree automaton A, is infected by M takes $O\left(|\tau|^{n\cdot(n+1)/2} \times |A| \times |A_M|\right)$ time and space.

Detection complexity in Theorem 5 is linear in the size of the tree automaton A_M which recognizes the set of traces reliably infected by M. This is an improvement on the exponential complexity bound of [7]. In practice, such an automaton contains traces whose arguments must cover $\mathcal{F}_d$, which may result in an increased complexity of the automaton. However, A_M is only used to restrict abstractions of traces from L to reliably infected traces, so only particular instances of the reliably infected traces are actually considered. Therefore, it seems wise in practice to represent A_M in a compact form, e.g. by defining constraints on the transitions, and to only instantiate variables when intersecting A_M with $R^{\leq n}(L)$. We do not detail this approach as it is implementation related and as it does not impact the worst-case complexity. Note that the same remarks hold for the tree transducer τ realizing the abstraction.

6 Application to Information Leak Detection

As we said at the beginning of this paper, abstraction can be applied to the detection of generic threats like sensitive information leak. Such a leak can be decomposed in two steps: capturing sensitive information and sending this information on the network. Data capture can be modelled by behavior patterns describing generic scenarios, e.g. capturing keystrokes or reading a passwords file, or by behavior patterns describing more ad hoc scenarios, e.g. reading data at a sensitive network location. For instance, keyboard capture can be modelled by a behavior pattern associated to unary function symbol $\lambda_{kb\,capture}$ whose argument represents the captured data:

$$\begin{aligned}\lambda_{kb\,capture}(x) := {} & GetAsyncKeyState(x) \vee \\ & (RegisterRawInputDevices(\texttt{GENERIC\_KBD}, \texttt{SINK}) \\ & \odot GetRawInputData(x, \texttt{INPUT})) \vee \\ & (\exists y.\, SetWindowsHookEx(y, \texttt{WH\_KEYBOARD\_LL}) \wedge \\ & \neg UnhookWindowsHookEx(y)\, \mathbf{U}\, \_HookCalled(y, x)).\end{aligned}$$

Note that the call $\_HookCalled$ is not strictly speaking a library call but denotes the execution of the hook callback function, which can be captured.

As for leaking data, it can be carried out via the network, a removable device, etc. For instance, the following behavior pattern describes the sending of data over the network, where x represents the sent data, y represents the recipient: $\lambda_{send}(x, y) := \exists z.\, sendto(z, x, y) \vee (connect(z, y) \wedge \neg close(z)\, \mathbf{U} send(z, x))$.

Data capture and data leak could also be realized by two other behavior patterns: a sensitive file reading behavior pattern associated to the unary function symbol $\lambda_{read\,secret}$, whose parameter represents the read data, and a removable

device copy behavior pattern associated to the binary function symbol $\lambda_{cp\,to\,rm}$, where the first argument represents the read data and y represents the removable device. We then define the data leak behavior by:
$dataleak\,(x,y) := (\lambda_{kb\,capture}\,(x) \vee \lambda_{read\,secret}\,(x)) \odot (\lambda_{send}\,(x,y) \vee \lambda_{cp\,to\,rm}\,(x,y))\,.$

Thus we detect information leak in a generic way. Moreover we can easily extend the definition of this behavior to other capture and leak scenarios.

Note that a difficulty of analysis lies at the dataflow level. At execution time, parameters of two functions may be directly related (by pointing to the same object) or indirectly related (the parameter of the second function may be a pointer to some field of the structure used by the first function, or it may be a copy of the parameter of the first function). Although an interesting extension of this formalism could be the formalization of this relation, we assume this relation is given at capture time.

References

1. Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Behavior Abstraction in Malware Analysis. In Oleg Sokolsky Grigore Rosu, editor, *1st International Conference on Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 168–182, St. Julians Malta, August 2010. Springer-Verlag.
2. J. Bergeron, M. Debbabi, J. Desharnais, MM. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *Symposium on Requirements Engineering for Information Security*, 2001.
3. Fred Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6(1):22–35, 1987.
4. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.
5. Rémi Gilleron and Sophie Tison. Regular Tree Languages and Rewrite Systems. *Fundamenta Informaticae*, 24:157–176, 1995.
6. J. Guttman, A. Herzog, J. Ramsdell, and C. Skorupta. Verifying information flow goals in security-enhanced linux. *J. Computer Security*, 13(1):115–134, 2005.
7. Grégoire Jacob, Hervé Debar, and Eric Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *International Symposium on Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2009.
8. Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2005.
9. Baudouin Le Charlier, Abdelaziz Mounji, and Morton Swimmer. Dynamic detection and classification of computer viruses using general behaviour patterns. In *International Virus Bulletin Conference*, pages 1–22, 1995.
10. Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. A layered architecture for detecting malicious behaviors. In *International symposium on Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 78–97. Springer, 2008.

11. J. Morales, P. Clarke, Y. Deng, and G. Kibria. Characterization of virus replication. *Journal in Computer Virology*, 4(3):221–234, August 2007.
12. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155. IEEE Computer Society, 2001.
13. Prabhat K. Singh and Arun Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *Information Assurance Workshop*, pages 298–300. IEEE Press, 2003.
14. A. Tanenbaum, J. Herder, and H. Nos. Can we make operating systems reliable and secure? *IEEE Computer*, pages 44–51, 2006.

A Additional Background

A.1 Term Algebras

A closed 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:

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

We define in a natural way the projection on an alphabet $\Sigma' \subseteq \Sigma \cup \Gamma$ of a term t of $T_{TRACE}(\mathcal{F}, X)$, where X is a set of variables of sort $DATA$, and we denote it by $\pi_{\Sigma'}(t)$ or, equivalently, by $t|_{\Sigma'}$. Similarly, the concatenation of two terms t and t' in $T_{TRACE}(\mathcal{F}, X)$, where X is a set of S-sorted variables and $t \notin X$, is denoted by $t \cdot t' \in T_{TRACE}(\mathcal{F}, X)$ 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 $2^{T_{TRACE}(\mathcal{F},X)} \times 2^{T_{ACTION}(\mathcal{F},X)}$ with $L \cdot L' = L \cdot \{a \cdot \epsilon \mid a \in L'\}$ and to $2^{T_{TRACE}(\mathcal{F},X)} \times T_{ACTION}(\mathcal{F}, X)$ with $L \cdot a = L \cdot \{a \cdot \epsilon\}$.

A.2 LTL Temporal Logic

Let A be an alphabet. We denote by A^ω the set of infinite words over A: $A^\omega = \{a_1 a_2 \ldots \mid \forall i,\, a_i \in A\}$.

Let AP be the set of atomic propositions. An LTL formula is as follows:

- $\top$ (true) and $\bot$ (false) are LTL formulas ;
- If $p \in AP$, then p is an LTL formula;
- If φ_1 and φ_2 are LTL formulas, then: $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\mathbf{X}\varphi_1$ ("next time"), $\mathbf{F}\varphi_1$ ("eventually" ou "in the future") and $\varphi_1 \,\mathbf{U}\, \varphi_2$ ("until") are LTL formulas.

A formula is satisfied on infinite sequences of sets of atomic predicates, denoted by $\xi = (a_0, a_1, \ldots) \in \left(2^{AP}\right)^\omega$. We denote by ξ^i the sequence $(a_i, a_{i+1}, \ldots)$. $\xi \models \varphi$ (ξ validates φ) is defined by:

- $\xi \models \top$;
- $\xi \models p$, where $p \in AP$, iff $p \in a_0$;
- $\xi \models \neg\varphi$ iff $\xi \not\models \varphi$;
- $\xi \models \varphi_1 \wedge \varphi_2$ iff $\xi \models \varphi_1$ and $\xi \models \varphi_2$;
- $\xi \models \varphi_1 \vee \varphi_2$ iff $\xi \models \varphi_1$ or $\xi \models \varphi_2$;
- $\xi \models \mathbf{X}\varphi$ iff $\xi^1 \models \varphi$;
- $\xi \models \mathbf{F}\varphi$ iff for some $i \geq 0$, $\xi^i \models \varphi$;
- $\xi \models \varphi_1 \mathbf{U} \varphi_2$ iff for some $i \geq 0$, $\xi^i \models \varphi_2$ and, for any $j \in [0..i-1]$, $\xi^j \models \varphi_1$.

Examples of closed FOLTL formulas are:

- $fopen \odot fwrite$;
- $\exists x, y.\, fopen(x) \odot \top \mathbf{U} fwrite(x, y)$;
- $\exists x, y.\, fopen(x) \odot \neg fclose(x) \,\mathbf{U}\, fwrite(x, y)$;
- $\exists x.\, fopen(x) \odot (\forall y.\, \neg fwrite(x, y)) \,\mathbf{U}\, fclose(x)$.

A.3 Tree Automata

Let X be a set of variables. A top-down tree automaton is a tuple $\mathcal{A} = (\mathcal{F}, Q, \Delta, q_0, Q_f)$ where $\mathcal{F}$ is an alphabet, Q is a finite set of states, $q_0 \in Q$ is an initial state, $Q_f \subseteq Q$ is a set of final states and Δ is a set of rules of the form:

$$q\left(f\left(x_1, \ldots, x_n\right)\right) \to f\left(q_1\left(x_1\right), \ldots, q_n\left(x_n\right)\right).$$

where $f \in \mathcal{F}$ of arity $n \in \mathbb{N}$, $q, q_1, \ldots, q_n \in Q$ and $x_1, \ldots, x_n \in X$. The transition relation $\to_{\mathcal{A}}$ with respect to $\mathcal{A}$ is defined by:

$$\begin{gathered}
\forall t, t' \in T\left(\mathcal{F} \cup Q\right), \\
t \to_{\mathcal{A}} t' \\
\Leftrightarrow \\
\exists q\left(f\left(x_1, \ldots, x_n\right)\right) \to f\left(q_1\left(x_1\right), \ldots, q_n\left(x_n\right)\right) \in \Delta, \\
\exists p \in Pos\left(t\right), \exists u_1, \ldots, u_n \in T\left(\mathcal{F}\right), \\
t|_p = q\left(f\left(u_1, \ldots, u_n\right)\right) \text{ and } t' = t\left[f\left(q_1\left(u_1\right), \ldots, q_n\left(u_n\right)\right)\right]_p
\end{gathered}$$

The language recognized by $\mathcal{A}$ is defined by: $\mathcal{L}\left(\mathcal{A}\right) = \{t \mid q_0\left(t\right) \to_{\mathcal{A}}^* q, q \in Q_f\}$. The tree languages recognized by top-down tree automata are the regular tree languages.

The size of $\mathcal{A}$ is defined by: $|\mathcal{A}| = |Q| + |\Delta|$.

A.4 Tree Transducers

Let X be a set of variables. A bottom-up tree transducer is a tuple $\tau = (\mathcal{F}, Q, Q_f, \Delta)$ where $\mathcal{F}$ is the finite set of input and output symbols, Q is a finite set of unary states, $Q_f \subseteq Q$ is the set of final states, Δ is a set of transduction rules of the form:

$$f\left(q_1\left(x_1\right), \ldots, q_n\left(x_n\right)\right) \to q\left(u\right)$$

where $f \in \mathcal{F}$ of arity $n \in \mathbb{N}$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in X$, $u \in T\left(\mathcal{F}, \{x_1, \ldots, x_n\}\right)$, or

$$q\left(x\right) \to q'\left(u\right) \quad (\epsilon\text{-rule})$$

where $q, q' \in Q$, $x \in X$, $u \in T\left(\mathcal{F}, \{x_1\}\right)$.

The transition relation $\to_\tau$ for the transducer τ is defined by:

$$\begin{gathered}
\forall t, t' \in T\left(\mathcal{F} \cup Q\right), \\
t \to_\tau t' \\
\Leftrightarrow \\
\exists f\left(q_1\left(x_1\right), \ldots, q_n\left(x_n\right)\right) \to q\left(u\right) \in \Delta, \\
\exists p \in Pos\left(t\right), \exists u_1, \ldots, u_n \in T\left(\mathcal{F}\right), \\
t|_p = f\left(q_1\left(u_1\right), \ldots, q_n\left(u_n\right)\right) \text{ and } t' = t\left[q\left(u\left\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\right\}\right)\right]_p
\end{gathered}$$

ϵ-rules are a particular case of this definition.

The transduction rule induced by τ is the relation R_τ defined by:$R_\tau = \{(t, t') \mid t \to_\tau^* q\left(t'\right), t \in T\left(\mathcal{F}\right), t' \in T\left(\mathcal{F}\right), q \in Q_f\}$. A bottom-up tree transducer

is linear if its rules are linear. A binary relation on $T_{TRACE}(\mathcal{F}, X)$ is called rational iff there exists a linear bottom-up tree transducer realizing it.

The size of τ is defined by: $|\tau| = |Q| + |\Delta|$.

The image of a regular set in $T_{TRACE}(\mathcal{F}, X)$ by a linear bottom-up tree transducer is a regular set in $T_{TRACE}(\mathcal{F}, X)$. Bottom-up tree transductions are closed by union, intersection and composition.

B Proofs

Theorem 6. *Let R be an abstraction relation, such that R and R^{-1} are rational. There exists a detection procedure deciding if $L \models M$, for any regular set of traces L and for any regular malicious behavior M having the property of (n, m)-completeness for some positive integers n and m.*

Proof. Let's define $M' = \pi_\Gamma^{-1}(T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma))$, where π_Γ^{-1} is the inverse of the projection on Γ. (n, m)-completeness of M can be restated as:

$$\begin{gathered} L \models M \\ \Leftrightarrow \\ \exists t' \in R^{\leq n}(L),\, R^{\leq n}(t') \subseteq M'. \end{gathered}$$

Let's show that the right member of this equivalence is decidable.

Observe first that, for any set $A \subseteq T_{TRACE}(\mathcal{F})$, any term $t \in T_{TRACE}(\mathcal{F})$ and any integer $i \in \mathbb{N}$, t can be rewritten by R into some term of A in i steps iff some term of A can be rewritten by R^{-1} into t in i steps:

$$R^i(t) \cap A \neq \emptyset \Leftrightarrow t \in \left(R^{-1}\right)^i(A) \qquad (1)$$

Hence:

$$\begin{gathered} R^{\leq n}(t') \subseteq M' \\ \Leftrightarrow \\ \neg\left(R^{\leq n}(t') \cap (T_{TRACE}(\mathcal{F}) \setminus M') \neq \emptyset\right) \\ \Leftrightarrow \\ \neg\left(\bigvee_{0 \leq i \leq n} \left(R^i(t') \cap (T_{TRACE}(\mathcal{F}) \setminus M')\right) \neq \emptyset\right) \\ \Leftrightarrow \\ \text{by (1)} \\ \neg\left(\bigvee_{0 \leq i \leq n} t' \in \left(R^{-1}\right)^i (T_{TRACE}(\mathcal{F} \setminus M'))\right) \\ \Leftrightarrow \\ \neg\left(t' \in \left(R^{-1}\right)^{\leq n} (T_{TRACE}(\mathcal{F} \setminus M'))\right) \end{gathered}$$

Intuitively, this set $\left(R^{-1}\right)^{\leq n}\left(T_{TRACE}\left(\mathcal{F}\setminus M'\right)\right)$ represents the set of unreliably infected traces to eschew. Let's denote by M'' its complement: $M'' = T_{TRACE}\left(\mathcal{F}\right)\backslash\left(R^{-1}\right)^{\leq n}\left(T_{TRACE}\left(\mathcal{F}\right)\setminus M'\right)$. The property of (n,m)-completeness can be restated as follows:

$$\begin{gathered} L \models M \\ \Leftrightarrow \\ R^{\leq n}\left(L\right) \cap M'' \neq \emptyset \end{gathered}$$

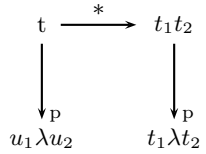
R^{-1} is rational so it preserves regularity. Also, M' is regular by hypothesis, so M'' is regular too. Similarly, R is rational and L is regular, so $R^{\leq n}\left(L\right)$ is regular too, hence the decidability of detection.

Theorems XX rely notably on a lemma stating that, whenever some behavior pattern is abstracted within a trace t after any number of steps, it can be abstracted from t in one step and at the same concrete position.

Definition 12 (Concrete Position). *Let t be a term of $T_{TRACE}\left(\mathcal{F}\right)$ and t' be a subterm of t, of sort $TRACE$. The concrete position of t' in t is the position of $t'|_{\Sigma}$ in $t|_{\Sigma}$.*

Definition 13 (Abstraction at a Concrete Position). *Let B be a behavior pattern associated to an abstraction symbol λ and equipped with an abstraction relation $\rightarrow$. We say that the trace $t = t_1 \cdot t_2$ is abstracted with respect to B into $t_1 \cdot \lambda \cdot t_2$ at the concrete postion p, denoted by $t_1 \cdot t_2 \rightarrow_p t_1 \cdot \lambda \cdot t_2$, iff $t_1 \cdot t_2 \rightarrow t_1 \cdot \lambda \cdot t_2$ and p is the concrete postion of t_2 in t.*

This lemma can therefore be stated as follows. If $t \rightarrow^* t_1 \cdot t_2 \rightarrow_p t_1 \cdot \lambda \cdot t_2$, then there exists $u_1, u_2 \in T\left(\mathcal{F}\right)$ such that: $t \rightarrow_p u_1 \cdot \lambda \cdot u_2$.



We actually show a more general form of this lemma, where a variable number of behavior patterns (not necessarily distinct) are abstracted one after the other.

Lemma 1. *Let $t \in T_{TRACE}\left(\mathcal{F}\right)$ be a trace and $\lambda_1, \lambda_2, \ldots \lambda_n \in T_{ACTION}\left(\mathcal{F}_\Gamma\right)$ be abstract actions. Let an abstraction chain from t be $t \rightarrow^* t_1 \cdot t'_1 \rightarrow_{p_1} t_1 \cdot \lambda_1 \cdot t'_1 \rightarrow^* t_2 \cdot t'_2 \rightarrow_{p_2} t_2 \cdot \lambda_2 \cdot t'_2 \rightarrow^* \ldots \rightarrow^* t_n \cdot t'_n \rightarrow_{p_n} t_n \cdot \lambda_n \cdot t'_n$ where we distinguish n abstraction steps, then:*

$$\begin{gathered} \exists u_1, \ldots u_n, u'_1, \ldots u'_n \in T_{TRACE}\left(\mathcal{F}\right), \\ t \rightarrow_{p_1} u_1 \cdot \lambda_1 \cdot u'_1 \rightarrow_{p_2} u_2 \cdot \lambda_2 \cdot u'_2 \rightarrow_{p_3} \ldots \rightarrow_{p_n} u_n \cdot \lambda_n \cdot u'_n \end{gathered}$$

Proof. By induction on the length of the derivation $t \rightarrow^* t_{n+1} \cdot t'_{n+1}$

- For the base case $k = 1$, we have: $t \rightarrow_{p_1} t_1 \cdot \lambda_1 \cdot t'_1$. Hence: $\exists u_1, u'_1,\ u_1 = t_1, u'_1 = t'_1$.
- For the induction step $n \Rightarrow n+1$. Assume the property for $l = n$. We prove the property for $l = n+1$. By the induction hypothesis applied to $t \rightarrow^* t_1 \cdot t'_1 \rightarrow_{p_1} t_1 \cdot \lambda_1 \cdot t'_1 \rightarrow^* t_2 \cdot t'_2 \ldots \rightarrow t_n \cdot t'_n \rightarrow_{p_n} t_n \cdot \lambda_n \cdot t'_n \rightarrow^* t_{n+1} \cdot t'_{n+1}$, $\exists u_1, \ldots u_n, u'_1, \ldots, u'_n \in T_{TRACE}(\mathcal{F}),\ t \rightarrow_{p_1} u_1 \cdot \lambda_1 \cdot u'_1 \rightarrow \ldots \rightarrow u_n \cdot \lambda_n \cdot u'_n$. We want to rewrite $u_n \cdot \lambda_n \cdot u'_n$ in $u_{n+1} \cdot \lambda_{n+1} \cdot u'_{n+1}$.
  For $l = n+1$, the chain of length n is extended by $t_n \cdot \lambda_n \cdot t'_n \rightarrow^* t_{n+1} \cdot t'_{n+1} \rightarrow t_{n+1} \cdot \lambda_{n+1} \cdot t'_{n+1}$.
  Now, existence of the reduction $t_{n+1} \cdot t'_{n+1} \rightarrow t_{n+1} \cdot \lambda_{n+1} \cdot t'_{n+1}$ entails the existence of an occurrence of the behavior pattern B_{n+1} in $t_{n+1} \cdot t'_{n+1}$. This occurrence also appears in $u_n \cdot \lambda_n \cdot u'_n$ and can therefore be abstracted at the same concrete postion p_{n+1}, hence the existence of terms u_{n+1} and u'_{n+1} such that: $u_n \cdot \lambda_n \cdot u'_n \rightarrow_{p_{n+1}} u_{n+1} \cdot \lambda_{n+1} \cdot u'_{n+1}$.

Theorem 7. *Let X be a set of variables of sort $DATA$. Let $\alpha_1, \ldots, \alpha_n \in T_{ACTION}(\mathcal{F}_\Gamma, X)$.*

The malicious behavior $M := \exists X.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_n$ has the property of $(n, 0)$-completeness.

Proof. Let $L \subseteq T(\mathcal{F}_\Sigma)$ be a set of traces.

$\Rightarrow$ By definition of the infection, there exists a trace $t \in L$ with a normal form $t\!\downarrow$ such that $t\!\downarrow|_\Gamma$ is in $T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$. Thereby, $t\!\downarrow$ can be written:

$$t\!\downarrow = t_1 \cdot \alpha_1 \cdot t_2 \cdots \alpha_n \cdot t_n$$

where $t_1, \ldots, t_n \in T_{TRACE}(\mathcal{F})$.

By Lemma 1, there exists $u_1, \ldots, u_n \in T_{TRACE}(\mathcal{F})$ such that t is abstracted into $t' = u_1 \cdot \alpha_1 \cdot u_2 \cdots \alpha_n \cdot u_n$ in exactly n steps. Thus $t' \in R^{\leq n}(L)$. Moreover $t'|_\Gamma \in T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$. Therefore, any future abstraction of t' will still contain this occurrence of M, hence: $R^*(t')|_\Gamma \subseteq T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$.

$\Leftarrow$ Let $t' \in R^{\leq n}(L)$ be a partial abstraction of a trace of L such that $R^{\leq n}(t')\big|_\Gamma \subseteq T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$. In particular, $t'|_\Gamma$ is in $T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$ so t' can be written $t' = t_1 \cdot \alpha_1 \cdot t_2 \cdots \alpha_n \cdot t_n$, where $t_1, \ldots, t_n \in T_{TRACE}(\mathcal{F})$. Clearly, any future abstraction of t' will still contain this occurrence of M and this will be especially true for its normal form $t'\!\downarrow \in L\!\downarrow_R$. Hence $t'\!\downarrow|_\Gamma \in T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$ and thus $L \models M$.

Theorem 8. *, $\alpha_3 = \lambda_3(\overline{x_3})$ be abstract actions such that X is exactly the set of variables appearing in $\overline{x_1}$ and $\overline{x_3}$ and variables of $\overline{x_2}$ are variables from $X \cup Y$.*

The malicious behavior $M := \exists X \forall Y, .\, \alpha_1 \wedge (\neg\alpha_2\, \mathbf{U}\, \alpha_3)$ has the property of $(3, 2)$-completeness.

Proof. Let $L \subseteq T(\mathcal{F}_\Sigma)$ be a set of traces.

$\Leftarrow$ Let $t' \in R^{\leq 3}(L)$ be a trace such that $R^{\leq 3}(t')\big|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$.

In particular, $t'|_\Gamma \in T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$ so there exists a substitution $\sigma_X : X \to T_{DATA}(\mathcal{F})$ such that we may decompose t' in $t' = u \cdot \alpha_1\sigma_X \cdot v \cdot \alpha_3\sigma_X \cdot w$, and such that no instance of $\alpha_2\sigma_X$ appears in v. We can choose σ_X, u, v and w in such a way that the malicious behavior does not occur in $\alpha_1\sigma_X \cdot v$ nor $v \cdot \alpha_3\sigma_X$:

$$\{\alpha_1\sigma_X \cdot v|_\Gamma\,,\ v|_\Gamma \cdot \alpha_3\sigma_X\} \notin T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)\,.$$

Let t'' be a normal form of t': $t'' \in \{t'\}\downarrow_R$. Assume L is not infected by M. Then $t''|_\Gamma \notin T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$ and there must exist a substitution $\sigma_Y : Y \to T_{DATA}(\mathcal{F}_d)$ such that the abstract action $\alpha_2\sigma_X\sigma_Y$ has been inserted at a concrete postion p between $\alpha_1\sigma_X$ and $\alpha_3\sigma_X$. By Lemma 1, we could have inserted this action $\alpha_2\sigma_X\sigma_Y$ directly in term t', at the same concrete postion p:

$$\exists t_1, t_2,\ t' \to_p t_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot t_2.$$

Considering that the insertion is made between actions $\alpha_1\sigma_X$ and $\alpha_3\sigma_X$, and given that $t' = u \cdot \alpha_1\sigma_X \cdot v \cdot \alpha_3\sigma_X \cdot w$, we can decompose v in $v = v_1 \cdot v_2$ such that insertion occurs after v_1, in other words:

$$t' \to_p u \cdot \alpha_1\sigma_X \cdot v_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot v_2 \cdot \alpha_3\sigma_X \cdot w.$$

Let's denote by t'_1 the obtained term: $t'_1 = u \cdot \alpha_1\sigma_X \cdot v_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot v_2 \cdot \alpha_3\sigma_X \cdot w$.

By hypothesis, $R^{\leq 3}(t')\big|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$. Yet, $t'_1 \in R^{\leq 3}(t')$ so there existed another occurrence of M in t'. Furthermore, $t' \in R^{\leq 3}(L)$ so at most three abstraction symbols appear within t'. Finally, by hypothesis, neither $\alpha_1\sigma_X \cdot v$ nor $v \cdot \alpha_3\sigma_X$ contain an abstract occurrence of the malicious behavior so we are necessarily in one of the following cases:

- There exists a substitution $\sigma'_X : X \to T_{DATA}(\mathcal{F}_d)$ such that $\alpha_1\sigma'_X$ appears in u, $\alpha_3\sigma'_X = \alpha_3\sigma_X$ and $\alpha_2\sigma_X \neq \alpha_2\sigma'_X$;
- There exists a substitution $\sigma'_X : X \to T_{DATA}(\mathcal{F}_d)$ such that $\alpha_3\sigma'_X$ appears in w, $\alpha_1\sigma'_X = \alpha_1\sigma_X$ and $\alpha_2\sigma_X \neq \alpha_2\sigma'_X$.

Both cases being symmetrical, let's assume we are in the first case.

The reasoning is similar: by non infection hypothesis, normal forms of t'_1 are not in $T(\mathcal{F}) \cdot \pi_\Gamma^{-1}(M) \cdot T(\mathcal{F})$ so an action $\alpha_2\sigma'_X\sigma'_Y$ can be inserted by abstraction between $\alpha_1\sigma'_X$ and $\alpha_3\sigma'_X$, for some substitution $\sigma'_Y : Y \to T_{DATA}(\mathcal{F})$. And by Lemma 1, this abstraction can be performed at the same concrete postion in t'_1, yielding a term $t'_2 \in R^2(t')$.

Yet, by the hypothesis $t' \in R^{\leq 3}(L)$, t' can not contain any other abstract action than the three actions previously identified ($\alpha_1\sigma'_X$, $\alpha_1\sigma_X$ and $\alpha_3\sigma_X$) so t'_2 does not contain any other occurrence of M. This contradicts hypothesis $R^{\leq 3}(t')\big|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$.

$\Rightarrow$ By definition of the infection, there exists a trace $t \in L$ such that one of its normal forms $t\downarrow$ is in $T(\mathcal{F}) \cdot \pi_\Gamma^{-1}(M) \cdot T(\mathcal{F})$ and can therefore be written:

$$t\downarrow = u \cdot \alpha_1\sigma_X \cdot v \cdot \alpha_3\sigma_X \cdot w$$

where $\sigma_X : X \to T_{DATA}(\mathcal{F})$ is a closed substitution over X, $u, v, w \in T_{TRACE}(\mathcal{F})$ and no instance of $\alpha_2\sigma_X$ appears in v.

We define a term $t' \in R^{\leq 3}(t)$ that can be written $t' = u' \cdot \alpha_1\sigma_X \cdot v' \cdot \alpha_3\sigma_X \cdot w'$, such that v' does not contain any instance of $\alpha_2\sigma_X$, and we show that no rewriting step from t' can make the trace leave the malicious behavior, that is no rewriting step from t' inserts an instance of $\alpha_2\sigma_X$ in v'. u', v' and w' are defined as follows:

- If $w = w_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot w_2$ where $w_1 \in T(\mathcal{F}_\Gamma)$ and $\sigma_Y : Y \to T_{DATA}(\mathcal{F})$ is a substitution over Y, then, by Lemma 1:

$$\exists u', v', w'_1, w'_2 \in T(\mathcal{F}_\Sigma),\ t \to\to u' \cdot \alpha_1\sigma_X \cdot v' \cdot \alpha_3\sigma_X \cdot w'_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot w'_2.$$

  We then define: $w' = w'_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot w'_2$.
- Otherwise, by Lemma 1:

$$\exists u', v', w' \in T(\mathcal{F}_\Sigma),\ t \to\to u' \cdot \alpha_1\sigma_X \cdot v' \cdot \alpha_3\sigma_X \cdot w'.$$

Clearly, v' does not contain any instance of $\alpha_2\sigma_X$ since the only abstract actions appearing in t' come from the application of Lemma 1. Hence, $t'|_\Gamma \in T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$.

We now show that: $R^+(t')|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$ (note we are not restricted to order 3 of abstraction). Assume this is not the case and that t' can thus be rewritten in such a way that an action $\alpha_2\sigma_X\sigma'_Y$ is inserted within v' for some substitution $\sigma'_Y : Y \to T(\mathcal{F})$. The occurrence of the behavior pattern responsible for this insertion must also appear in $t\downarrow$. However, $t\downarrow$ is in normal form so this occurrence has already been abstracted, at the same concrete postion, that is after a concrete action de v. Moreover, by hypothesis no instance of $\alpha_2\sigma_X$ appears in v since $t\downarrow$ is in $T(\mathcal{F}) \cdot \pi_\Gamma^{-1}(M) \cdot T(\mathcal{F})$ so action $\alpha_2\sigma_X\sigma'_Y$ necessarily appears in the abstract actions at the head of w. But this corresponds to the first case of the two previous cases, so this occurrence had already been abstracted in t'. Since the abstraction is terminating, this occurrence can not anymore be abstracted, hence a contradiction.

Hence: $R^*(t')|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$.

Theorem 9. *Let B be a behavior pattern and R be an abstraction relation with respect to B defined from an abstraction system composed of rules $A_i(X) \cdot B_i(X) \cdot y \to A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X) \cdot y$ such that the set $\bigcup_{i\in[1..n]} \bigcup_{\sigma\in Subst_X} A_i(X)\sigma \cdot \lambda(\overline{x})\sigma \cdot B_i(X)\sigma$ is recognized by a tree automaton A_R. Then R and R^{-1} are rational and realized by two transducers of size $O(|A_R|)$.*

Proof. We construct bottom-up linear tree transducers realizing R and R^{-1}.

Let n be the number of rules of the abstraction system. Let C denote the set $\bigcup_{i\in[1..n]} \bigcup_{\sigma\in Subst_X} A_i(X)\sigma \cdot \lambda(\overline{x})\sigma \cdot B_i(X)\sigma \subseteq T_{TRACE}(\mathcal{F})$.

C is regular and there exists a bottom-up linear tree transducer $\tau_C = (\mathcal{F}, Q, Q_f, \Delta)$ of size $O(|A_R|)$ realizing the identity over $T_{TRACE}(\mathcal{F}_\Sigma) \cdot C \cdot T_{TRACE}(\mathcal{F}_\Sigma)$, such

that rules of Δ are of the form $f\left(q_1\left(x_1\right), \ldots, q_n\left(x_n\right)\right) \rightarrow q\left(f\left(x_1, \ldots, x_n\right)\right)$, where $f \in \mathcal{F}$ of arity $n \in \mathbb{N}$, and such that every state is reachable and co-reachable (i.e. reachable from a final state), i.e.: $\forall q \in Q, \exists t \in T\left(\mathcal{F}\right), t \rightarrow_{\tau_C}^* q\left(t\right)$ and $\forall q \in Q, \forall t \in T_{TRACE}\left(\mathcal{F}\right), \exists u \in T_{TRACE}\left(\mathcal{F}\right), \exists p \in Pos\left(u\right), \exists q_f \in Q_f$, $t \rightarrow_{\tau_C}^* u\left[q\left(t\right)\right]_p \rightarrow_{\tau_C}^* q_f\left(u\right)$.

In the case of the abstraction relation R, we construct a set of states Q' and a set of rules Δ' in the following way:

- Initially, $Q' = Q$, $\Delta' = \Delta$;
- When the abstraction symbol λ is read on the input (and written on the output), we modify rules so that it is still written on the output but is not anymore read on the input (note that the $A_i\left(X\right)$ and $B_i\left(X\right)$ sets are composed of concrete traces that do not contain the abstraction symbol λ). For each rule $\lambda\left(q_1\left(x_1\right), \ldots, q_n\left(x_n\right)\right) \rightarrow q_\lambda\left(\lambda\left(x_1, \ldots, x_n\right)\right)$ in Δ and for each rule $\cdot\left(q_\lambda\left(x\right), q_B\left(x'\right)\right) \rightarrow q_{\lambda B}\left(\cdot\left(x, x'\right)\right)$ in Δ:
  - **Remove** rule $\cdot\left(q_\lambda\left(x\right), q_B\left(x'\right)\right) \rightarrow q_{\lambda B}\left(\cdot\left(x, x'\right)\right)$ from Δ' (i.e. we cannot anymore reach $q_{\lambda B}$ after reading λ);
  - **Create** a state q_α in Q';
  - For any term $\alpha = \lambda\left(d_1, \ldots, d_n\right)$, $d_1, \ldots, d_n \in \mathcal{F}_d$, such that $\alpha \rightarrow_\Delta^* q_\lambda\left(\alpha\right)$, **add** to Δ' a rule $q_B\left(x\right) \rightarrow q_\alpha\left(\cdot\left(\alpha, x\right)\right)$ (i.e. when state q_B is reached, write the abstract action on the output);
  - For any rule $\cdot\left(q_a\left(x\right), q_{\lambda B}\left(x'\right)\right) \rightarrow q_{a\lambda B}\left(\cdot\left(x, x'\right)\right)$, **add** to Δ' a rule $\cdot\left(q_a\left(x\right), q_\alpha\left(x'\right)\right) \rightarrow q_{a\lambda B}\left(\cdot\left(x, x'\right)\right)$ (since we used to reach $q_{\lambda B}$ by going through state q_B and reading the abstract action while we know directly write the abstract action when reaching q_B and go into q_α).
- **Create** generic states $q_{\mathcal{F}_d}$ and q_Γ in Q respectively consuming any data and any abstract action, and **add** to Δ' the corresponding rules: $d \rightarrow q_{\mathcal{F}_d}\left(d\right)$ for every data symbol $d \in \mathcal{F}_d$ and $f\left(q_{\mathcal{F}_d}\left(x_1\right), \ldots, q_{\mathcal{F}_d}\left(x_n\right)\right) \rightarrow q_\Gamma\left(f\left(x_1, \ldots, x_n\right)\right)$ for every abstract action symbol $f \in \Gamma$ or arity $n \in \mathbb{N}$;
- Authorize interleaving with any abstract action, except at the end of the terms of $A_i\left(X\right)\sigma$: for all state q in Q' which is not one of the states q_α we created previously, **add** to Δ' a rule $\cdot\left(q_\Gamma\left(x_1\right), q\left(x_2\right)\right) \rightarrow q\left(\cdot\left(x_1, x_2\right)\right)$.

Denote by $\tau = \left(\mathcal{F}, Q', Q_f, \Delta'\right)$ the bottom-up linear tree transducer just defined. Let's show that τ realizes R.

$R \subseteq R_\tau$ Assume $\left(u \cdot v, u \cdot \alpha \cdot v\right) \in R$. By Definition 4 of an abstraction relation, $u = u' \cdot a$ with $a \in T_{ACTION}\left(\mathcal{F}_\Sigma\right)$. The term $u|_\Sigma \cdot \alpha \cdot v|_\Sigma$ is in C and is therefore read and output unmodified by τ_C. In other words, there exists states $q_B, q_\lambda, q_{\lambda B}, q_{a\lambda B} \in Q$ et $q_f \in Q$ such that:

$$\begin{aligned} u'|_\Sigma \cdot a \cdot \alpha \cdot v|_\Gamma &\rightarrow_{\tau_C}^* u'|_\Sigma \cdot a \cdot q_\lambda\left(\alpha\right) \cdot q_B\left(v|_\Sigma\right) \\ &\rightarrow_{\tau_C} u'|_\Sigma \cdot a \cdot q_{\lambda B}\left(\alpha \cdot v|_\Sigma\right) \\ &\rightarrow_{\tau_C}^* u'|_\Sigma \cdot q_{a\lambda B}\left(a \cdot \alpha \cdot v|_\Sigma\right) \\ &\rightarrow_{\tau_C}^* q_f\left(u'|_\Sigma \cdot a \cdot \alpha \cdot v|_\Sigma\right) \end{aligned}$$

Assume there exists $b \in T_{ACTION}(\mathcal{F}_\Sigma)$ such that $b \to^*_{\tau_C} q_\lambda(b)$. Then, we would have: $u'|_\Sigma \cdot a \cdot b \cdot v|_\Gamma \to^*_{\tau_C} q_f(u'|_\Sigma \cdot a \cdot b \cdot v|_\Gamma)$ and thus $u'|_\Sigma \cdot a \cdot b \cdot v|_\Gamma$ would be in C which is impossible.

Since the only rules removed from Δ are rules of the form $\cdot(q_\lambda(x), q_B(x')) \to q_{\lambda B}(\cdot(x, x'))$, we still have in τ: $v|_\Sigma \to^*_\tau q_B(v|_\Sigma)$ and $\alpha \to^*_\tau q_\lambda(\alpha)$. Similarly, for $w \in T_{TRACE}(\mathcal{F})$ and $q, q' \in Q$, if $u'|_\Sigma \cdot q(w) \to^*_\tau q'(u'|_\Sigma \cdot w)$, without going through any state q_α, then: $u' \cdot q(w) \to^*_\tau q'(u' \cdot w)$.

By construction of τ, given that for any state q different from a state q_α, we have a rule $\cdot(q_\Gamma(x_1), q(x_2)) \to q(\cdot(x_1, x_2))$, we infer: $v \to^*_\tau q_B(v)$. Still by constructions of τ, we have $q_B(v) \to_\tau q_\alpha(\alpha \cdot v)$ and then $a \cdot q_\alpha(\alpha \cdot v) \to^*_\tau q_{a\lambda B}(a \cdot \alpha \cdot v)$ and finally $u' \cdot q_{a\lambda B}(a \cdot \alpha \cdot v) \to^*_\tau q_f(u' \cdot a \cdot \alpha \cdot v)$. We conclude: $u \cdot v \to^*_\tau q_f(u \cdot \alpha \cdot v)$.

$R_\tau \subseteq R$ Assume now that $u \to^*_\tau q_f(v)$, where $q_f \in Q_f$. By construction of τ, only abstract actions were added along the transduction so u and v can be written: $u = u_0 \cdots u_n$ and $v = u_0 \cdot \alpha_1 \cdot u_1 \cdots \alpha_n \cdot u_n$, where $u_0, \ldots, u_n \in T_{TRACE}(\mathcal{F})$ and $\alpha_1, \ldots, \alpha_n \in T_{ACTION}(\mathcal{F}_\Gamma)$.

Before going further, let's observe that we have the following results:

- For all $u \in T_{TRACE}(\mathcal{F})$, if $u \to^*_\tau q(u)$ with $q \in Q$ then $u|_\Sigma \to^*_{\tau_C} q(u|_\Sigma)$, since the term has been output unmodified so no state q_α was run across and new rules in $\Delta' \setminus \Delta$ have only been applied to read intermediate abstract actions.
- Similarly, for all $u, u' \in T_{TRACE}(\mathcal{F})$, if $u \cdot q(u') \to^*_\tau q'(u \cdot u')$ with $q, q' \in Q$ then $u|_\Sigma \cdot q(u') \to^*_{\tau_C} q'(u|_\Sigma \cdot u')$.

We show by induction on n that:

$$\begin{gathered}
\forall u_0, \ldots, u_n \in T_{TRACE}(\mathcal{F}), \forall \alpha_1, \ldots, \alpha_n \in T_{ACTION}(\mathcal{F}_\Gamma), \forall q \in Q \\
u_0 \cdots u_n \to^*_\tau q(u_0 \cdot \alpha_1 \cdot u_1 \cdots \alpha_n \cdot u_n) \\
\Downarrow \\
u_0|_\Sigma \cdot \alpha_1 \cdot u_1|_\Sigma \cdots \alpha_n \cdot u_n|_\Sigma \to^*_{\tau_C} q(u_0|_\Sigma \cdot \alpha_1 \cdot u_1|_\Sigma \cdots \alpha_n \cdot u_n|_\Sigma) \\
\text{and} \\
\forall i \in [1..n], u_{i-1} \in T_{TRACE}(\mathcal{F}) \cdot T_{ACTION}(\mathcal{F}_\Sigma)
\end{gathered}$$

$n = 0$ This corresponds to the first of the previous intermediary results.

$n \Rightarrow n+1$ By construction of τ, there exists $q_B \in Q$ and $q_\alpha \in Q' \setminus Q$ such that:

$$\begin{aligned}
u_0 \cdots u_{n+1} &\to^*_\tau u_0 \cdot q_B(u_1\alpha_2 \cdots \alpha_{n+1}u_{n+1}) \\
&\to_\tau u_0 \cdot q_\alpha(\alpha_1 \cdot u_1\alpha_2 \cdots \alpha_{n+1}u_{n+1}) \\
&\to^*_\tau q(u_0 \cdot \alpha_1 \cdot u_1\alpha_2 \cdots \alpha_{n+1}u_{n+1})
\end{aligned}$$

By induction hypothesis applied to $u_1 \cdots u_{n+1} \to^*_\tau q_B(u_1\alpha_2 \cdots \alpha_{n+1}u_{n+1})$: $u_1|_\Sigma \cdot \alpha_2 \cdots \alpha_{n+1} \cdot u_{n+1}|_\Sigma \to^*_{\tau_C} q_B(u_1|_\Sigma \cdot \alpha_2 \cdots \alpha_{n+1} \cdot u_{n+1}|_\Sigma)$.

Let's define $v = u_1|_\Sigma \cdot \alpha_2 \cdots \alpha_{n+1} \cdot u_{n+1}|_\Sigma$. The induction hypothesis then entails:

$$v \to^*_{\tau_C} q_B(v) \tag{1}$$

Moreover, we have:

$$\begin{aligned} u_0 \cdot q_B(v) &\to_\tau u_0 \cdot q_\alpha(\alpha_1 \cdot v) \\ &\to_\tau^* q(u_0 \cdot \alpha_1 \cdot v) \end{aligned}$$

By construction, there exists a rule $\cdot(q_\lambda(x), q_B(x')) \to q_{\lambda B}(\cdot(x, x'))$ in Δ such that $\alpha_1 \to_{\tau_C}^* q_\lambda(\alpha_1)$. We deduce:

$$\alpha_1 \cdot q_B(v) \to_{\tau_C} q_{\lambda B}(\alpha_1 \cdot v) \tag{2}$$

$q_\alpha \notin Q$ so there exists a rule $\cdot(q_a(x), q_{\lambda B}(x')) \to q_{a\lambda B}(\cdot(x, x'))$ in Δ and the associated rule $\cdot(q_a(x), q_\alpha(x')) \to q_{a\lambda B}(\cdot(x, x'))$ in Δ' such that:

- u_0 can be written: $u_0 = u_0' \cdot a$, where $u_0' \in T_{TRACE}(\mathcal{F})$ and $a \in T_{ACTION}(\mathcal{F})$.
- and $a \to_{\tau_C}^* q_a(a)$.

Thus, in τ_C :

$$a \cdot q_{\lambda B}(\alpha_1 \cdot v) \to_{\tau_C}^* q_{a\lambda B}(a \cdot \alpha_1 \cdot v) \tag{3}$$

and, in τ :

$$u_0' \cdot a \cdot q_\alpha(\alpha_1 \cdot v) \to_\tau^* u_0' \cdot q_{a\lambda B}(a \cdot \alpha_1 \cdot v) \to_\tau^* q(u_0 \cdot \alpha_1 \cdot v) \tag{4}$$

Finally, by combining (2) and (3): $a \cdot \alpha_1 \cdot q_B(v) \to_{\tau_C}^* q_{a\lambda B}(a \cdot \alpha_1 \cdot v)$.

By applying the second of the previous intermediary results to (4), we also have: $u_0'|_\Sigma \cdot q_{a\lambda B}(a \cdot \alpha_1 \cdot v) \to_{\tau_C}^* q(u_0'|_\Sigma \cdot a \cdot \alpha_1 \cdot v)$.

By combing (1) with the previous results, we get: $u_0|_\Sigma \cdot \alpha_1 \cdot v \to_{\tau_C}^* q(u_0|_\Sigma \cdot \alpha_1 \cdot v)$.

Finally, assume a is abstract. a is recognized by τ_C in q_a. Moreover, by construction, terms recognized by τ_C in $q_{\lambda B}$ start with an abstract action so, by the rule $\cdot(q_a(x), q_{\lambda B}(x')) \to q_{a\lambda B}(\cdot(x, x'))$, τ_C would recognize in $q_{a\lambda B}$ a term containing two abstract actions. Considering that $q_{a\lambda B}$ is co-reachable, and that terms in C only contain one abstract action, this is impossible. So a is indeed in Σ.

Conclude by observing that terms in C only contain one abstract action, namely an instance of $\lambda(\overline{x})$, so only the case $n = 1$ is possible. Since u_0 ends with a concrete action, we have: $(u_0 \cdot u_1, u_0 \cdot \alpha_1 \cdot u_1) \in R$.

As for the size of τ, observe that τ has $O(|Q|)$ states and $O(|\Delta| + |\mathcal{F}|)$ transition rules.

In the case of the inverse abstraction relation R^{-1}, the construction is the same, except for the abstraction symbol which must be deleted instead of being inserted. Thus, instead of adding a rule $q(x) \to q_\alpha(\cdot(\alpha, x))$ to Δ' for every $\alpha = \lambda(d_1, \ldots, d_n)$ recognized in q_λ, we add a single rule $\cdot(q_\lambda(x), q(x')) \to q_\alpha(x')$.

Lemma 2. *Let τ be a bottom-up linear tree transducer and let A be a top-down tree automaton. $\tau(\mathcal{L}(A))$ is a regular tree language recognized by a top-down tree automaton of size $O(|\tau| \times |A|)$.*

Proof. Let τ_A be the bottom-up tree transducer recognizing the identity on A. τ_A has the same size as A. Let's denote by $\tau' = \tau_A \circ \tau$ transduction τ restricted to $\mathcal{L}(A)$. τ' has a size $O(|\tau_A| \times |\tau|)$ and there exists an automaton A' which recognizes the image of τ' and has the same size as τ'. Moreover, it precisely recognizes $\tau(\mathcal{L}(A))$.

Lemma 3. *Let R be an abstraction relation. Let M be a regular malicious behavior with the property of (n, m)-completeness for some positive integers n and m.*

If R^{-1} is rational, then the set of traces reliably infected by M with respect to R is regular.

Proof. The set $T_{TRACE}(\mathcal{F}) \setminus \left(T_{TRACE}(\mathcal{F}) \cdot \pi_\Gamma^{-1}(M) \cdot T_{TRACE}(\mathcal{F})\right)$ is regular since inverse projection, concatenation and complementation preserve regularity.

Sets $\left(R^{-1}\right)^i \left(T_{TRACE}(\mathcal{F}) \setminus \left(T_{TRACE}(\mathcal{F}) \cdot \pi_\Gamma^{-1}(M) \cdot T_{TRACE}(\mathcal{F})\right)\right)$ are regular by Lemma 2, as is their union for $1 \leq i \leq m$ the complement of their union.

Theorem 10. *Let R be an abstraction relation such that R and R^{-1} are rational. Let τ be a bottom-up tree transducer realizing R. Let M be a regular malicious behavior with the property of (n, m)-completeness and A_M be a tree automaton recognizing the set of traces reliably infected by M with respect to R.*

Deciding if a regular set of traces L, recognized by a tree automaton A, is infected by M takes $O\left(|\tau|^{n\cdot(n+1)/2} \times |A| \times |A_M|\right)$ time and space.

Proof. Let's denote by M'' the set of traces reliably infected by M with respect to R. The proof of Theorem 4 relied on the following result:

$$\begin{gathered} L \models M \\ \Leftrightarrow \\ R^{\leq n}(L) \cap M'' = \emptyset \end{gathered}$$

By Lemma 2, $R^{\leq n}(L)$ has a size $O\left(|\tau|^{n\cdot(n+1)/2} \times |A|\right)$. Intersection of two tree automata A_1 and A_2 yields an automaton of size $O(|A_1| \times |A_2|)$. Finally, deciding if an automaton recognizes the empty set takes time and space linear in its size.

C Examples

Example 5. Let B be a behavior pattern constructed from the sequence $a(x) \cdot b \cdot c(x)$ such that the sequence $d(x) \cdot e(x)$ frees x and is therefore forbidden between $a(x)$ and $c(x)$.

Let's define:

$$T_{a_1 \cdots a_n}(\mathcal{F}) = T_{TRACE}(\mathcal{F}) \cdot a_1 \cdot T_{TRACE}(\mathcal{F}) \cdots a_n \cdot T_{TRACE}(\mathcal{F}).$$

We then define B by:

$$B = \bigcup_{\sigma} \left(\left(a\left(x\right) \cdot T_b\left(\mathcal{F}\right) \cdot c\left(x\right) \right) \sigma \setminus T_{(d(x) \cdot e(x))\sigma}\left(\mathcal{F}\right) \right).$$

Assume action b effectively realizes the behavior pattern functionality: abstraction with respect to this pattern then corresponds to inserting the abstraction symbol λ immediately after action b.

In order to define the set of rewriting rules which compose the abstraction system, we need to consider the case where the sequence $d\left(x\right) \cdot e\left(x\right)$ appears between $a\left(x\right)$ and b or between b and $c\left(x\right)$, and the case where action $d\left(x\right)$ appears between $a\left(x\right)$ and b and action $e\left(x\right)$ appears between b and $c\left(x\right)$. Thus, we define three rewriting rules using the following $A_i\left(x\right)$ and $B_i\left(x\right)$ sets:

- $A_1\left(x\right) = a\left(x\right) \cdot T\left(\mathcal{F}\right) \setminus T_{d(x)}\left(\mathcal{F}\right) \cdot b$;
- $B_1\left(x\right) = T\left(\mathcal{F}\right) \setminus T_{e(x)}\left(\mathcal{F}\right) \cdot c\left(x\right)$;
- $A_2\left(x\right) = a\left(x\right) \cdot T_{d(x)}\left(\mathcal{F}\right) \setminus T_{d(x) \cdot e(x)}\left(\mathcal{F}\right) \cdot b$;
- $B_2\left(x\right) = T\left(\mathcal{F}\right) \setminus T_{e(x)}\left(\mathcal{F}\right) \cdot c\left(x\right)$;
- $A_3\left(x\right) = a\left(x\right) \cdot T\left(\mathcal{F}\right) \setminus T_{d(x)}\left(\mathcal{F}\right) \cdot b$;
- $B_3\left(x\right) = T_{e(x)}\left(\mathcal{F}\right) \setminus T_{d(x) \cdot e(x)}\left(\mathcal{F}\right) \cdot c\left(x\right)$.

Importance of the choice of the insertion position for the abstraction symbol is illustrated by the following example.

Consider a behavior pattern describing the reading of a sensitive file $ReadFile$ and a behavior pattern describing the sending of data over the network $socket \cdot sendto$. The trace $socket \cdot ReadFile \cdot sendto$ will be deemed suspicious only when the abstraction symbol identifying the reading of a sensitive file is inserted immediately after $ReadFile$ and the abstraction symbol identifying the sending of data over the network is inserted after $sendto$. Indeed, in that case the trace will be interpreted as the the reading of a sensitive file followed by a network communication. The choice of this insertion position therefore allows the detection algorithm to reduce false positives and false negatives.