# Abstraction by Term Rewriting for Malware Behavior Analysis – Extended Version –

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 execution traces with respect to given behavior patterns in order to produce a high level representation of a program behavior and then, by comparing this abstract form to signatures defining reference abstract malicious behaviors. Abstraction is performed by term rewriting using rules on terms with variables, which enables to handle the data used by behavior functionalities. This technique allows us to deal with interleaved behaviors. Successfully applied to malware detection, it allows us in particular to model and detect information 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 data of interest. As a result, part of the program behavior remains unknown. In a static approach, the program code is analyzed without being executed. With this approach, a more exhaustive representation of the program can be constructed.

However, when we try to analyze a binary code, which is nowadays practically always the case in malware detection, the program semantics is not available. So it is not possible to directly perform usual static analysis. Determining the program semantics is an intractable problem, for two main reasons. First, there are indirect jumps, like the `jmp eax` instruction which jumps to the value pointed by the x86-register `eax`. Second, a program may use complex code protection, for example by dynamically modifying 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 as complete as possible, 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 [4] 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 [11,14,13]. This approach is quite limited, which motivated some recent works. In [8], the authors use attribute automata, at the price of an exponential time complexity detection. Model-checking is used in [3,9,15] to track data. But none of these works considers functional polymorphism. Moreover they do not tackle either the problem of constructing a high-level view of a program behavior, which limits their applicability. In [12], functional polymorphism is considered by preprocessing execution traces and transforming them into a high-level representation capturing their semantic meaning. But as this approach deals with the execution trace being observed, it analyzes a single behavior at a time.

In this paper, we propose an approach allowing program analysis by abstracting behavior components in program traces. Rather than working on single traces, we consider unbounded sets of traces, that may come, as said previously, from runtime analysis or from static or dynamic analysis. Thus the constructed representation of a program behavior is more complete than we could do with a single trace.

This abstraction amounts to identifying a set of known functionalities described by behavior patterns, and to rewriting these behavior patterns into abstract functionality symbols. The abstract form of an execution trace is thus defined in terms of these functionalities and not anymore in terms of observed actions, which are low-level and thus less reliable. This allows us to deal with the problem of functional polymorphism. Then, we can detect in linear time whether a program exhibits a given behavior. Finally, our formalism is particularly adapted to the protection against generic threats like the leak of sensitive information.

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 current and future variants of this behavior. 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 predefined malicious behaviors.

**Analysis of suspicious 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 behavioral similarity with other malware, etc.

*Road map* 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 or packing techniques, etc.

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

the functionality of capturing keystrokes by the formula:

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

with two elementary actions *RegisterRawInputDevices* and *GetRawInputData*, 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 action $GetAsyncKeyState$ and thus we complete the formula $\varphi_{kb\,capture}$ by adding a disjunction with $\exists x.\, GetAsyncKeyState\,(x)$.

Then, we can directly express a malicious behavior in terms of the functionalities instead of elementary actions. Here, it is $\varphi_{kb\,capture}$ followed by $\varphi_{write\,file}$.

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

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

The trace $t$ is analyzed in terms of functionalities. A subtrace of $t$ validates the formula $\varphi_{kb\,capture}$ and therefore carries out the functionality of capturing keystrokes. Similarly, a subtrace of $t$ validates the formula $\varphi_{write\,file}$ and therefore carries out the functionality of writing to a file.

Moreover, these functionalities are not malicious in themselves but their combination is. This combination can be seen as a component of a reference malicious behavior $M$. Thus, we are able to detect a malicious behavior by identifying in a trace combinations of the functionalities composing the behavior. In our example, $t$ is malicious, not because it separately validates $\varphi_{kb\,capture}$ and $\varphi_{write\,file}$, but because it validates the combination of both functionalities in the order $\varphi_{kb\,capture}$ followed by $\varphi_{write\,file}$.

Detection of a behavior is therefore performed on traces abstracted at the level of functionalities, that is at a higher level than the level of elementary actions. To that end, we represent functionalities by abstract symbols in a set $\Gamma$ and we define a behavior on $\Gamma$. We then need to define an abstraction relation $R_\alpha$ transforming a raw trace into an abstract trace on $\Gamma$. This relation is defined with a rewriting system whose left-hand sides of rules precisely identify occurrences of functionalities. A trace $t$ then exhibits the behavior $M$ if its abstract form contains an occurrence of $M$:

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

For instance, using the previous functionalities, we associate a unary function symbol $\lambda_{kb\,capture}$ in $\Gamma$ to the keystroke capture functionality $\varphi_{kb\,capture}$ and a binary function symbol $\lambda_{write\,file}$ in $\Gamma$ to the file write functionality $\varphi_{write\,file}$.

We then define the information leak behavior by the set of terms:

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

Note the importance of parameters of abstraction symbols to describe the data they manipulate.

*Previous works* In [2], we already proposed to abstract program trace languages with respect to behavior patterns, for detection and analysis. But patterns were defined by 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 the rewritten ones.

For instance, a behavior pattern identifying a sequence $a \cdot b$ was associated to the string rewriting system rewriting any string $a \cdot u \cdot b$ into $\lambda$. So, when defining a new behavior pattern by the sequence $c \cdot d$ associated to a symbol $\lambda'$, the trace $a \cdot c \cdot d \cdot b$ was abstracted either into $\lambda$, or into $\lambda'$, but never into $\lambda \cdot \lambda'$ or $\lambda' \cdot \lambda$.

The goal of this paper is then 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 object;
- to give parameters to behavior patterns themselves in order to analyze the dataflow in abstracted traces.

Another main difference with the previous result [2] is that we now can detect information leaks i.e., we can prevent unauthorized disclosure or modifications of information. Indeed, the technique we propose here allows us to dynamically track some data flow, as we just saw it. Here again, our point of view is that dynamic analysis may be complementary to static analysis and formal methods. Although verification is now applied to critical modules of systems, using test methods, model-checking, or formal proofs, which is necessary to obtain trusted systems with fine grained compartmentalization, this is not sufficient to 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 [7] which implements mandatory access control on Linux – but we should pay the price of security, as suggested by Tanenbaum [16].

## 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 distinct and:

- $\mathcal{F}_t = \{\epsilon, \cdot\}$ is the set of the trace constructors;

- $\mathcal{F}_a$ is a finite set of function symbols or constants, with signature $DATA^n \rightarrow ACTION$, $n \in \mathbb{N}$, describing actions;
- $\mathcal{F}_d$ is a finite set of constants of type $DATA$, describing 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 $\epsilon$ 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 $\sigma$ to a term $t \in T(\mathcal{F}, X)$ and by $L\sigma$ the application of $\sigma$ to a set of terms $L \subseteq T(\mathcal{F}, X)$. The *set of ground substitutions* over $X$ is denoted by $Subst_X$.

We partition $\mathcal{F}_a$ in a set $\Sigma$ of symbols denoting concrete actions and a set $\Gamma$ 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 concatenation* and *the projection* 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 representation of a program is chosen to be its set of traces. When executing a program, the captured data is represented on the alphabets $\Sigma$ and $\mathcal{F}_d$. In this paper, we consider that the captured data is the library calls along with their arguments. $\Sigma$ therefore represents the finite set of library calls, while 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 $T_{TRACE}(\mathcal{F}_\Sigma)$. 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 subset of the First-Order Linear Temporal Logic defined in [10], without the equality predicate. 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 $\varphi$ is an LTL formula, then $\varphi$ is an FOLTL formula ;
- If $\varphi$ 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: $\forall Y.\varphi \equiv \neg\exists Y.\neg\varphi$.

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 ground substitution over $Y$. The *application of* $\sigma$ to an FOLTL formula $\varphi$ is naturally defined by the formula $\varphi\sigma$ where any free variable $x$ in $\varphi$ 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 $\xi = (a_0, a_1, \ldots) \in \left(2^{AP}\right)^{\omega}$. $\xi \models \varphi$ ($\xi$ validates $\varphi$) is defined in the same way as for the LTL logic, with the additional rule: $\xi \models \exists Y.\varphi$ iff there exists a substitution $\sigma \in Subst_Y$ such that $\xi \models \varphi\sigma$.

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 finite 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 $\varphi$, denoted by $t \models \varphi$, iff $(\{a_0\}, \ldots, \{a_n\}, \{\}, \{\}, \ldots) \models \varphi$.

Notation $\varphi_1 \odot \varphi_2$ will stand for $\varphi_1 \wedge \mathbf{X}\varphi_2$. For examples, see the appendix.

***Tree Automata and Tree Transducers*** Tree automata and tree transducers are defined as usual (see appendix and [5]). A tree language is regular if it is recognized by some tree automaton and a binary relation is rational if it is realized by some tree transducer.

## 3 Trace Abstraction

The problem under study can be formalized in the following way. First, we 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 $R_\alpha$ 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 represented by 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 files, sending a mail or pinging a remote host. Such a functionality can be realized in different ways: by using different system calls, different library calls, different programming languages, etc.

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

*Example 1.* Let's consider the functionality of sending a ping. One way of realizing it consists in calling the *socket* function with the particular parameter `IPPROTO_ICMP` and then in calling the *sendto* function with the particular parameter `ICMP_ECHOREQ` describing the data to be sent. Between these two calls, the socket should not have been freed or reallocated. This is described by the FOLTL formula: $\exists x, y.\, socket\,(x, \alpha) \wedge (\neg closesocket\,(x)\ \mathbf{U}\ sendto\,(x, \beta, y) \wedge X\varphi_{end})$, where constants $\alpha$ and $\beta$ in $\mathcal{F}_d$ identify parameters `IPPROTO_ICMP` and `ICMP_ECHOREQ`, 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 parameter is the target, and the unique parameter of *closesocket* is the freed socket.

Formula $\varphi_{end}$ is a special formula defined by $\varphi_{end} := \bigwedge_{a \in T_{ACTION}(\mathcal{F}_\Sigma)} \neg a$, which indicates the end of the trace. This guarantees that only traces ending with the *sendto* action validate the formula. The idea is to only focus in the behavior pattern on the relevant actions of the functionality, by excluding irrelevant prefixes and suffixes.

A ping may also be realized using the function *IcmpSendEcho*, whose parameter represents the ping target. This corresponds to the FOLTL formula: $\exists x.\, IcmpSendEcho\,(x) \wedge X\varphi_{end}$.

Hence, the ping functionality may be described by the FOLTL formula:

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

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

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

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

*Example 2.* Using the FOLTL formula describing the ping functionality, in Example 1, the ping behavior pattern is defined as the set:

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

### 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 $\Gamma$, that we call abstraction symbol, 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 to define the order in which their functionalities were realized (see appendix for an example).

In addition, the abstraction symbol can have parameters corresponding to those used by the behavior pattern occurrence. This allows us to express dataflow constraints in a signature. For instance, the abstraction symbol for the ping behavior pattern could take a unique 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.

As said in the introduction, replacing a behavior pattern occurrence by its abstraction symbol precludes proper handling of interleaved behavior patterns occurrences, hence our choice here of preserving the occurrence when performing the abstraction.

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

When realizing the ping using the *socket* and *sendto* actions, the *sendto* action effectively performs the ping, so we wish to insert $\lambda_{ping}$ just after it and then to rewrite the trace $socket\,(1,\alpha)\cdot gethostbyname\,(2)\cdot sendto\,(1,\beta,3)\cdot closesocket\,(1)$ into the trace $socket\,(1,\alpha)\cdot gethostbyname\,(2)\cdot sendto\,(1,\beta,3)\cdot \lambda_{ping}\,(3)\cdot closesocket\,(1)$.

Abstraction of the ping in this case 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:

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

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

When realizing the ping using the *IcmpSendEcho* action, we want to insert the abstraction symbol $\lambda_{ping}$ after this action. Abstraction of the ping in this case therefore corresponds to rewriting a trace using 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 $B_2\,(x)=\{\epsilon\}$.

The abstraction relation is therefore defined 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 2 (Abstraction System).** *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 $T_{TRACE}(\mathcal{F})$ is a finite set of rewrite 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 rewrite rules we use generates a reduction relation on $T_{TRACE}(\mathcal{F})$ such that filtering works on traces projected on $\Sigma$ and the abstraction symbol is inserted after the last concrete action of a term in $A_i(X)$.

**Definition 3.** *The* reduction relation *on $T_{TRACE}(\mathcal{F})$ generated by a system with $n$ rewrite 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 $t, t' \in T_{TRACE}(\mathcal{F})$:*

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

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.

**Definition 4 (Abstraction).** *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 *with respect to this behavior pattern is the reduction relation on $T_{TRACE}(\mathcal{F})$ generated by an abstraction system composed of $n$ rules $A_i(X) \cdot B_i(X) \rightarrow A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ verifying:*

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

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 the abstraction relations with respect to each behavior pattern of $C$.*

From now on, if a behavior pattern is defined using an FOLTL formula $\varphi$ and associated to an abstraction symbol $\lambda$, we may describe it by the 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 trace. In the general case of a (possibly infinite) set of traces $L$, 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 the total abstraction $L{\downarrow_R}$ consisting in the set of normal forms of traces of $L$ 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 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 $\alpha$ is the inserted abstract action, then if $t_2$ starts with a sequence of abstract actions, $\alpha$ does not appear in this sequence.

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

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

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 $R^*(L)$. However, $R^*(L)$ is not computable in general [6], but only for some classes of term rewriting systems and when $L$ is regular [5]. Unfortunately, the term rewriting system implementing the abstraction relation does not belong to any of these classes. We will see later how to deal with this problem.

## 4 Detection Problem

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

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

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:*

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

In other words, the set of abstracted traces with respect to $R$ contains a subtrace with a behavior of $M$. This definition relies on the construction of the set $L{\downarrow_R}$, which is undecidable in general, as said above. 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 $\Gamma$ is in $M$, so we would wrongly infer that trace $t$ is malicious.

Example 4 shows 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 compromising detection i.e., traces seemingly exhibiting a malicious behavior though a few additional abstraction steps would make them leave the signature.

Consequently, we want to exclude unreliably infected traces 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 malicious behaviors in practice in the field of malware detection. This property states that, to show that a program is infected, a necessary and sufficient condition is that there exists a partially abstracted trace, abstracted in at most $m$ abstraction steps, that is infected and whose descendants up to the order $n$ are still infected.

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

**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 $n$, instead of $R^*(t)$.

From this observation, we suggest a new expression of infection where there are two orders of abstraction $m$ and $n$ such that $L$ is infected by $M$ if there is a trace $t' \in R^{\leq m}(L)$ which is reliably infected i.e., such that all descendants of $t'$ with respect to $R$ until the order $n$ are malicious.

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

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

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

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

We first prove that simple malicious behaviors describing sequences of abstract actions with no constraints other than dataflow constraints have the property of $(m, n)$-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 appendix C.

We now show that more complex malicious behaviors, expressing constraints on the abstract actions which appear interleaved with the malicious behavior have this property. An example of such malicious behaviors is: $\exists x \forall y.\, \lambda_1(x) \wedge \neg\lambda_2(x, y)\ \mathbf{U}\ \lambda_3$. For this purpose, we distinguish two sets of variables: existential variables represented by the set $X$ and universal variables represented by a set $Y$, which can only be used with negations.

**Theorem 2.** *Let* $X$*,* $Y$ *be two disjoint sets of variables of sort* $DATA$*. Let* $\lambda_1(\overline{x_1})$*,* $\lambda_2(\overline{x_2})$*,* $\lambda_3(\overline{x_3})$ *be abstract actions with* $\lambda_2 \neq \lambda_1$ *and* $\lambda_2 \neq \lambda_3$ *and 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* $M := \exists X \forall Y.\, \lambda_1(\overline{x_1}) \wedge (\neg\lambda_2(\overline{x_2})\ \mathbf{U}\ \lambda_3(\overline{x_3}))$ *has the property of* $(3, 2)$*-completeness.*

We expect these results to be generalized to more general forms of signatures. 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 1.* 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 traces realizing the behavior patterns are complex) and from the variables instantiations.

## 5 Rational Abstraction

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., we represent it by a tree automaton. Then, when it comes to effectively abstracting it i.e., to constructing a representation of its partially abstract form, the formalism of tree transducers is a suitable approach with interesting formal (closure of union, composition, preservation of regularity) and computational properties.

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 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 a terminating 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 linear bottom-up tree automaton $A_R$ without $\epsilon$-rules. Then $R$ and $R^{-1}$ are rational and realized by two linear bottom-up tree transducers of size $O(|A_R|)$.*

Then, we show that rationality of $R$ and $R^{-1}$ entails the decidability of detection.

**Theorem 4.** *Let $R$ be a terminating 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 $(m,n)$-completeness for some positive integers $m$ and $n$.*

When a malicious behavior $M$ has the property of $(m,n)$-completeness, the set of traces reliably infected by $M$ is the set defined as follows.

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

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

Using the set of traces $n$-reliably infected by $M$, 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 [8].

**Theorem 5.** *Let $R$ be a terminating abstraction relation such that $R$ and $R^{-1}$ are rational. Let $\tau$ be a linear bottom-up tree transducer realizing $R$. Let $M$ be a regular malicious behavior with the property of $(m,n)$-completeness and $A_M$ be a tree automaton recognizing the set of traces $n$-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 $O\left(|\tau|^{m\cdot(m+1)/2} \times |A| \times |A_M|\right)$ time and space.*

Note that in practice, the automaton $A_M$, recognizing the set of traces $n$-reliably infected by $M$, contains traces instantiated over the whole set $\mathcal{F}_d$. But $A_M$ is

only used to restrict, by intersection, $R^{\leq n}(L)$ to reliably infected traces. So only particular instances of the traces recognized by $A_M$ are actually considered. Therefore, it seems wise in practice to represent $A_M$ in a compact form, where arcs resulting from instantiations have been aggregated into arcs with variables, and to only instantiate these variables when intersecting $A_M$ with $R^{\leq n}(L)$. We do not detail this optimization as it is implementation related and it does not impact the worst-case complexity. Note that the same remarks hold for the tree transducer $\tau$ 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, and in particular to the detection of 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 the 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, as we would expect from the definition of $\Sigma$, but denotes the execution of the hook callback function, which can be captured.

As for data leak, 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 information 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.

We have partly implemented the abstraction formalism in a tool which uses Pin [1] for dynamic construction of the program behavior, with early positive and encouraging experimental results.

## References


1. Pin. http://www.pintool.org.
2. 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.
3. 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.
4. Fred Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6(1):22–35, 1987.
5. 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.
6. Rémi Gilleron and Sophie Tison. Regular Tree Languages and Rewrite Systems. *Fundamenta Informaticae*, 24:157–176, 1995.
7. 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.
8. 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.
9. 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.
10. Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. An EATCS Series. 2008.
11. Baudouin Le Charlier, Abdelaziz Mounji, and Morton Swimmer. Dynamic detection and classification of computer viruses using general behaviour patterns. In *Proceedings of the International Virus Bulletin Conference*, pages 1–22, 1995.
12. 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.
13. J. Morales, P. Clarke, Y. Deng, and G. Kibria. Characterization of virus replication. *Journal in Computer Virology*, 4(3):221–234, August 2007.
14. 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.

15. 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.
16. Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *IEEE Computer*, 39:44–51, May 2006.

# A Additional Background

## A.1 Term Algebras

A ground substitution on a finite set $X$ of $S$-sorted variables is a mapping $\sigma : X \to T(\mathcal{F})$ such that: $\forall s \in S, \forall x \in X_s, \sigma(x) \in T_s(\mathcal{F})$. $\sigma$ 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 $\epsilon$ 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 $A^\omega$ 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 $\varphi_1$ and $\varphi_2$ 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" or "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 $\xi^i$ the sequence $(a_i, a_{i+1}, \ldots)$. $\xi \models \varphi$ ($\xi$ validates $\varphi$) 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 $\Delta$ 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 $x_1, \ldots, x_n \in X$, or:

$$q \to q' \quad (\epsilon\text{-rule})$$

where $q, q' \in Q$.

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, $\Delta$ 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 $\tau$ is defined by:

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

$\epsilon$-rules are a particular case of this definition.

The transduction rule induced by $\tau$ is the relation $R_\tau$ defined by:$R_\tau = \{(t, t') \mid t \to_\tau^* q(t'), t \in T(\mathcal{F}), t' \in T(\mathcal{F}), q \in Q_f\}$. 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 $\tau$ 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 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:

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

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

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

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

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.

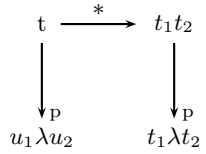
## C Proofs

Theorems 1 and 2 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 $T_{TRACE}(\mathcal{F})$ 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 $\lambda$ 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 position $p$, *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 position 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 $u_1, u_2 \in T(\mathcal{F})$ 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 $t \in T_{TRACE}(\mathcal{F})$ be a trace and $\lambda_1, \lambda_2, \ldots \lambda_n \in T_{ACTION}(\mathcal{F}_\Gamma)$ 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:*

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

*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 $t \rightarrow^* t_1 \cdot t'_1 \rightarrow_{p_1} t_1 \cdot \lambda_1 \cdot t'_1 \rightarrow^* t_2 \cdot t'_2 \ldots \rightarrow t_n \cdot t'_n \rightarrow_{p_n} t_n \cdot \lambda_n \cdot t'_n \rightarrow^* t_{n+1} \cdot t'_{n+1}$, $\exists u_1, \ldots u_n, u'_1, \ldots, u'_n \in T_{TRACE}(\mathcal{F})$, $t \rightarrow_{p_1} u_1 \cdot \lambda_1 \cdot u'_1 \rightarrow \ldots \rightarrow u_n \cdot \lambda_n \cdot u'_n$. For $l = n+1$, the chain of length $n$ is extended by $t_n \cdot \lambda_n \cdot t'_n \rightarrow^* t_{n+1} \cdot t'_{n+1} \rightarrow t_{n+1} \cdot \lambda_{n+1} \cdot t'_{n+1}$.
We want to rewrite $u_n \cdot \lambda_n \cdot u'_n$ in $u_{n+1} \cdot \lambda_{n+1} \cdot u'_{n+1}$.
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 position $p_{n+1}$, 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 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.*

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

$$L \models M \Leftrightarrow \exists t' \in R^{\leq n}(L),\ R^{\leq 0}(t')\big|_\Gamma \subseteq T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma).$$

$\Rightarrow$ By definition of the infection, there exists a trace $t \in L$ with a normal form $t\!\downarrow$ such that $t\!\downarrow\big|_\Gamma$ 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 $t'\big|_\Gamma \in T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$. Therefore, every future abstraction of $t'$ will still contain this occurrence of $M$, hence: $R^*(t')\big|_\Gamma \subseteq T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$.

$\Leftarrow$ Let $t' \in R^{\leq n}(L)$ be a partial abstraction of a trace of $L$ such that $R^0(t')\big|_\Gamma \subseteq T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$. So $t'\big|_\Gamma$ is in $T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$ and $t'$ 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 $t'\!\downarrow\big|_\Gamma \in T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma)$ and thus $L \models M$.

**Theorem 2.** *Let $X$, $Y$ be two disjoint sets of variables of sort DATA. Let $\lambda_1(\overline{x_1})$, $\lambda_2(\overline{x_2})$, $\lambda_3(\overline{x_3})$ be abstract actions with $\lambda_2 \neq \lambda_1$ and $\lambda_2 \neq \lambda_3$ and 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 $M := \exists X \forall Y. \lambda_1(\overline{x_1}) \wedge (\neg\lambda_2(\overline{x_2})\ \mathbf{U}\ \lambda_3(\overline{x_3}))$ has the property of $(3, 2)$-completeness.*

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

$$\begin{gathered} L \models M \\ \Leftrightarrow \\ \exists t' \in R^{\leq 3}(L),\ R^{\leq 2}(t')\big|_\Gamma \subseteq T_{TRACE}(\mathcal{F}_\Gamma) \cdot M \cdot T_{TRACE}(\mathcal{F}_\Gamma). \end{gathered}$$

$\Leftarrow$ Let $t' \in R^{\leq 3}(L)$ be a trace such that $R^{\leq 2}(t')\big|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$.

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 $\sigma_X$, $u$, $v$ and $w$ in such a way that the malicious behavior does not occur in $\alpha_1\sigma_X \cdot v$ nor $v \cdot \alpha_3\sigma_X$:

$$\{\alpha_1\sigma_X \cdot v|_\Gamma,\ v|_\Gamma \cdot \alpha_3\sigma_X\} \notin T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma). \tag{1}$$

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 $\sigma_Y \in Subst_Y$ such that the abstract action $\alpha_2\sigma_X\sigma_Y$ has been inserted at a concrete position $p$ 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 position $p$:

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

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, $R^{\leq 2}(t')\big|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$. Yet, $t'_1 \in R^{\leq 2}(t')$ so there exists an occurrence of $M$ in $t'_1$ and hence there existed another occurrence of $M$ in $t'$ than the one within which we have inserted $\alpha_2\sigma_X\sigma_Y$. Furthermore, $t' \in R^{\leq 3}(L)$ so at most three abstraction symbols appear within $t'$. Hence, the second occurrence of $M$ must share one of its actions with the first occurrence of $M$. Finally, by hypothesis (1), 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 two cases:

- There exists a substitution $\sigma'_X \in Subst_X$ 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 $\sigma'_X \in Subst_X$ 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.

We now reason on this occurrence of $M$ in $t'_1$ similarly to the way we did for the first occurrence of $M$ in $t'$.

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 $\sigma'_Y \in Subst_Y$. And by Lemma 1, this abstraction can be performed at the same concrete position in $t'_1$, yielding a term $t'_2 \in R^2(t')$, where the second occurrence of $M$ has been invalidated by the action $\alpha_2\sigma'_X\sigma'_Y$.

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 $R^{\leq 2}(t')\big|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$.

$\Rightarrow$ 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 $\sigma_X \in Subst_X$ is a ground substitution over $X$, $u, v, w \in T_{TRACE}(\mathcal{F})$ and no instance of $\alpha_2\sigma_X$ appears in $v$.

We first define a term $t' \in R^{\leq 3}(t)$ that contains an occurrence of $M$ i.e., 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 $\alpha_2\sigma_X$. The terms $u'$, $v'$ and $w'$ are defined as follows:

- If the abstract actions at the head of $w$ contain an instance of $\alpha_2\sigma_X$, that is if $w = w_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot w_2$ where $w_1 \in T(\mathcal{F}_\Gamma)$ and $\sigma_Y \in Subst_Y$, then, by Lemma 1:

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

  We then define: $w' = w'_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot w'_2$. Thus $t' = u' \cdot \alpha_1\sigma_X \cdot v' \cdot \alpha_3\sigma_X \cdot w'_1 \cdot \alpha_2\sigma_X\sigma_Y \cdot w'_2$. Moreover, since $t \in L$ is concrete, $v'$ contains no abstract action.
- Otherwise, by Lemma 1:

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

  Thus $t' = u' \cdot \alpha_1\sigma_X \cdot v' \cdot \alpha_3\sigma_X \cdot w'$. Moreover, since $t \in L$ is concrete, $v'$ contains no abstract action.

As we observed for each case, $v'$ contains no abstract action, so it contains no instance of $\alpha_2\sigma_X$. Hence, $t'|_\Gamma \in T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$.

We now show that: $R^{\leq 3}(t')\big|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$. For this, we show that: $R^*(t')|_\Gamma \subseteq T(\mathcal{F}_\Gamma) \cdot M \cdot T(\mathcal{F}_\Gamma)$. 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 $\sigma'_Y \in Subst_Y$. 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 position, that is after a concrete action of $v$.

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 has already been abstracted, which resulted in

the $\alpha_2\sigma_X\sigma_Y$ action in $t'$. As $R$ is a terminating abstraction relation, a behavior pattern occurrence can only be abstracted once, so this occurrence can not anymore be abstracted, which contradicts the assumption that an abstraction from $t'$ inserted this action $\alpha_2\sigma_X\sigma'_Y$.

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

We use the following definition and lemma to prove the rationality of abstraction.

**Definition 14.** *Let $\Omega$ be a set of function symbols with signature $DATA^n \to ACTION$, $n \in \mathbb{N}$. Let $L \subseteq T_{TRACE}(\mathcal{F})$ be a set of traces. The $\Omega$-generalized form of $L$ is the set:*

$$\Pi_\Omega(L) = \{t_0 \cdot a_1 \cdot t_1 \cdots a_n \cdot t_n \mid \begin{array}{l} t_0, \ldots, t_n \in T_{TRACE}(\mathcal{F}), \\ a_1, \ldots, a_n \in T_{ACTION}(\Omega \cup \mathcal{F}_d), \\ t_0 \cdots t_n \in L\}. \end{array}$$

**Lemma 2.** *Let $\Omega \subseteq \mathcal{F}_a$ be a set of function symbols with signature $DATA^n \to ACTION$, $n \in \mathbb{N}$. If $A$ is a bottom-up tree automaton without $\epsilon$-rules, then there exists a bottom-up tree automaton without $\epsilon$-rules of size $O(|A|)$ recognizing the $\Omega$-generalized form of $\mathcal{L}(A)$.*

*Proof.* The proof is straightforward since it amounts to adding loops consuming actions of $T_{ACTION}(\Omega \cup \mathcal{F}_d)$ on the states of $A$ that recognize terms of sort $TRACE$.

More specifically, let $A$ be an automaton $A = (\mathcal{F}, Q, Q_f, \Delta)$. We construct an automaton $A' = (\mathcal{F}, Q', Q_f, \Delta')$ in the following way:

- Initially, $Q' = Q$ and $\Delta' = \Delta$;
- Create generic states $q_{\mathcal{F}_d}$ and $q_\Omega$ in $Q'$ respectively consuming any data symbol in $\mathcal{F}_d$ and any action in $T(\Omega \cup \mathcal{F}_d)$, and add to $\Delta'$ the corresponding rules: $d \to q_{\mathcal{F}_d}(d)$ for every symbol $d \in \mathcal{F}_d$ and $f(q_{\mathcal{F}_d}(x_1), \ldots, q_{\mathcal{F}_d}(x_n)) \to q_\Omega(f(x_1, \ldots, x_n))$ for every function symbol $f \in \Omega$ of arity $n \in \mathbb{N}$;
- For each rule $\cdot(q(x), q'(x')) \to q''(\cdot(x, x')) \in \Delta$, add a rule $\cdot(q_\Omega(x), q''(x')) \to q''(\cdot(x, x'))$;
- For each rule $\epsilon \to q''(\epsilon)$, add a rule $\cdot(q_\Omega(x), q''(x')) \to q''(\cdot(x, x'))$.

We finally prove that: $\mathcal{L}(A') = \Pi_\Omega(\mathcal{L}(A))$.

Let $t \in T_{TRACE}(\mathcal{F})$. If $t$ is recognized by $A'$, then there is a derivation $t \to^*_{\Delta'} q_f(t)$ with $q_f \in Q_f$. During this derivation, the used rules of $\Delta'$ of the form $\cdot(q(x), q'(x')) \to q''(\cdot(x, x'))$ are either rules of $\Delta$ or new rules of the form: $\cdot(q_\Omega(x), q''(x')) \to q''(\cdot(x, x'))$. So, $t$ can be decomposed into: $t = t_0 \cdot a_1 \cdot t_1 \cdots a_n \cdot t_n$ such that:

$$\begin{array}{l} t \to^*_{\Delta'} t_0 \cdot a_1 \cdot t_1 \cdots a_n \cdot q_n(t_n) \\ \quad \to^*_{\Delta'} t_0 \cdot a_1 \cdot t_1 \cdots q_\Omega(a_n) \cdot q_n(t_n) \\ \quad \to^*_{\Delta'} t_0 \cdot a_1 \cdot t_1 \cdots q_n(a_n \cdot t_n) \\ \quad \to^*_{\Delta'} t_0 \cdot a_1 \cdot t_1 \cdots q_{n-1}(t_{n-1} \cdot a_n \cdot t_n) \\ \quad \cdots \\ \quad \to^*_{\Delta'} t_0 \cdot q_1(a_1 \cdot t_1 \cdots a_n \cdot t_n) \\ \quad \to^*_{\Delta'} q_0(t_0 \cdot a_1 \cdot t_1 \cdots a_n \cdot t_n) \end{array}$$

where $q_0, \dots q_n \in Q$, $q_0 = q_f$ and, for all $i \in [1..n]$, $a_i \rightarrow^*_{\Delta'} q_\Omega(a_i)$.

For all $i \in [1..n]$, existence of the reduction $t_{i-1} \cdot q_i(a_i \cdot t_i \cdots t_n) \rightarrow^*_{\Delta'} q_{i-1}(t_{i-1} \cdot a_i \cdot t_i \cdots t_n)$ entails that: $\forall u \in T_{TRACE}(\mathcal{F} \cup \Omega)$, $t_{i-1} \cdot q_i(u) \rightarrow^*_{\Delta'} q_{i-1}(t_{i-1} \cdot u)$.

Hence:

$$\begin{aligned} t_0 \cdots t_n &\rightarrow^*_{\Delta'} t_0 \cdot t_1 \cdots q_n(t_n) \\ &\rightarrow^*_{\Delta'} t_0 \cdot t_1 \cdots q_{n-1}(t_{n-1} \cdot t_n) \\ &\cdots \\ &\rightarrow^*_{\Delta'} t_0 \cdot q_1(t_1 \cdots t_n) \\ &\rightarrow^*_{\Delta'} q_0(t_0 \cdot t_1 \cdots t_n) \end{aligned}$$

Assume, without loss of generality, that the state $q_\Omega$ does not appear in the derivation $t_0 \cdots t_n \rightarrow^*_{\Delta'} q_f(t_0 \cdots t_n)$. Then the state $q_{\mathcal{F}_d}$ does not appear either in this derivation, so only rules of $\Delta$ are used. Hence, $t_0 \cdots t_n \rightarrow^*_{\Delta} q_f(t_0 \cdots t_n)$, so $t_0 \cdots t_n \in \mathcal{L}(A)$, and finally $t \in \mathrm{II}_\Omega(\mathcal{L}(A))$.

Conversely, if $t$ is in $\mathrm{II}_\Omega(\mathcal{L}(A))$, $t$ can be written: $t = t_0 \cdot a_1 \cdot t_1 \cdots a_n \cdot t_n$ with, for all $i$, $a_i \rightarrow^*_{\Delta'} q_\Omega(a_i)$ and $t_0 \cdots t_n \rightarrow^*_{\Delta} q_f(t_0 \cdots t_n)$ for some $q_f \in Q_f$. Hence, using the new rules, we have: $t \rightarrow^*_{\Delta'} q_f(t)$ and $t \in \mathcal{L}(A')$.

**Theorem 3.** *Let $B$ be a behavior pattern and $R$ be a terminating 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 bottom-up tree automaton $A_R$ without $\epsilon$-rules. Then $R$ and $R^{-1}$ are rational and realized by two linear bottom-up tree transducers of size $O(|A_R|)$.*

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

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

Let $\diamondsuit$ be a constant of sort $ACTION$ not in $\mathcal{F}_a$. We consider the following set:

$$\begin{aligned} Img_\diamondsuit(R) = \{ t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \mid\ & t_1, t_2 \in T_{TRACE}(\mathcal{F}), \\ & \alpha \in T_{ACTION}(\mathcal{F}_\Gamma), \\ & (t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R \}. \end{aligned}$$

We will show that this set is recognized by a bottom-up tree automaton without $\epsilon$-rules, of size $O(|A_R|)$, and then use this set to construct a tree transducer recognizing $R$ and $R^{-1}$.

Define $C'$ to be the set:

$$C' = \mathrm{II}_{\{\diamondsuit\}}(C) \cap (T_{TRACE}(\mathcal{F}) \cdot \diamondsuit \cdot T_{ACTION}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{TRACE}(\mathcal{F})).$$

By Lemma 2 applied to $\Omega = \{\diamondsuit\}$ and to automaton $A_R$, the set $\mathrm{II}_{\{\diamondsuit\}}(C)$ is regular and recognized by a bottom-up tree automaton without $\epsilon$-rules, of size $O(|A_R|)$. The set $T_{TRACE}(\mathcal{F}) \cdot \diamondsuit \cdot T_{ACTION}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{TRACE}(\mathcal{F})$ is also recognized by a bottom-up tree automaton, without $\epsilon$-rules, and of constant size. So their intersection $C'$ is regular and recognized by a bottom-up tree automaton $A_{C'}$ without $\epsilon$-rules, of size $O(|A_R|)$. Terms in $C'$ are terms of $C$ where the abstract action has been enclosed between two diamond symbols.

Now, define $C''$ to be the set

$$C'' = \amalg_{\Gamma}(C') \cap (T_{TRACE}(\mathcal{F}) \cdot T_{ACTION}(\mathcal{F}_{\Sigma}) \cdot \diamondsuit \cdot T_{ACTION}(\mathcal{F}_{\Gamma}) \cdot \diamondsuit \cdot T_{TRACE}(\mathcal{F})).$$

By Lemma 2 applied to $\Omega = \Gamma$ and to automaton $A_{C'}$, the set $\amalg_{\Gamma}(C')$ is regular and recognized by a bottom-up tree automaton without $\epsilon$-rules, of size $O(|A_R|)$. The set $T_{TRACE}(\mathcal{F}) \cdot T_{ACTION}(\mathcal{F}_{\Sigma}) \cdot \diamondsuit \cdot T_{ACTION}(\mathcal{F}_{\Gamma}) \cdot \diamondsuit \cdot T_{TRACE}(\mathcal{F})$ is also recognized by a bottom-up tree automaton, without $\epsilon$-rules, and of constant size. So their intersection $C''$ is regular and recognized by a bottom-up tree automaton $A_{C''}$ without $\epsilon$-rules, of size $O(|A_R|)$.

Finally, we remove from $C''$ the terms violating the terminating condition of Definition 6. We define $C'''$ to be the set:

$$C''' = C'' \setminus \bigcup_{\alpha \in T_{ACTION}(\mathcal{F}_{\Gamma})} T_{TRACE}(\mathcal{F}) \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot T_{TRACE}(\mathcal{F}_{\Gamma}) \cdot \alpha \cdot T_{TRACE}(\mathcal{F}).$$

The set $T_{ACTION}(\mathcal{F}_{\Gamma})$ is finite, so the set $\bigcup_{\alpha \in T_{ACTION}(\mathcal{F}_{\Gamma})} T_{TRACE}(\mathcal{F}) \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot T_{TRACE}(\mathcal{F}_{\Gamma}) \cdot \alpha \cdot T_{TRACE}(\mathcal{F})$ is recognized by a tree automaton of constant size. Hence, $C'''$ is regular, recognized by a bottom-up tree automaton without $\epsilon$-rules, of size $O(|A_{C''}|) = O(|A_R|)$, and it verifies:

$$T_{TRACE}(\mathcal{F}) \cdot C''' \cdot T_{TRACE}(\mathcal{F}) = Img_{\diamondsuit}(R). \tag{2}$$

Indeed, for all $t_1, t_2 \in T_{TRACE}(\mathcal{F}), \alpha \in T_{ACTION}(\mathcal{F}_{\Gamma})$:

$$\begin{gathered}
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R \\
\Leftrightarrow \\
\exists u_1, v_1, u_2, v_2 \in T_{TRACE}(\mathcal{F}),\ t_1 = u_1 \cdot v_1, t_2 = v_2 \cdot u_2, \\
v_1|_{\Sigma} \cdot \alpha \cdot v_2|_{\Sigma} \in C, v_1 \in T_{TRACE}(\mathcal{F}) \cdot T_{ACTION}(\mathcal{F}_{\Sigma}), \\
v_2 \notin T_{TRACE}(\mathcal{F}_{\Gamma}) \cdot \alpha \cdot T_{TRACE}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1, u_2, v_2 \in T_{TRACE}(\mathcal{F}),\ t_1 = u_1 \cdot v_1, t_2 = v_2 \cdot u_2, \\
v_1|_{\Sigma} \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot v_2|_{\Sigma} \in C', v_1 \in T_{TRACE}(\mathcal{F}) \cdot T_{ACTION}(\mathcal{F}_{\Sigma}), \\
v_2 \notin T_{TRACE}(\mathcal{F}_{\Gamma}) \cdot \alpha \cdot T_{TRACE}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1, u_2, v_2 \in T_{TRACE}(\mathcal{F}),\ t_1 = u_1 \cdot v_1, t_2 = v_2 \cdot u_2, \\
v_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot v_2 \in C'', \\
v_2 \notin T_{TRACE}(\mathcal{F}_{\Gamma}) \cdot \alpha \cdot T_{TRACE}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1, u_2, v_2 \in T_{TRACE}(\mathcal{F}),\ t_1 = u_1 \cdot v_1, t_2 = v_2 \cdot u_2, \\
v_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot v_2 \in C''' \\
\Leftrightarrow \\
t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \in T_{TRACE}(\mathcal{F}) \cdot C''' \cdot T_{TRACE}(\mathcal{F}).
\end{gathered}$$

We now define the transducers realizing $R$ and $R^{-1}$. To that end, let's consider the relation $T$ defined by:

$$T = \{(t \cdot t', t \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t') \,, \mid t, t' \in T_{TRACE}(\mathcal{F}) \,, \alpha \in T_{ACTION}(\mathcal{F}_\Gamma)\} \,.$$

Clearly, relations $T$ and $T^{-1}$ are rational and recognized by transducers $\tau_T$ and $\tau_{T^{-1}}$ of constant size.

The set $T_{TRACE}(\mathcal{F}) \cdot C''' \cdot T_{TRACE}(\mathcal{F})$ is recognized by a bottom-up tree automaton without $\epsilon$-rules, of size $O(|A_R|)$, so there exists a linear bottom-up tree transducer $\tau_{C'''}$ realizing the relation $\{(t,t) \mid t \in T_{TRACE}(\mathcal{F}) \cdot C''' \cdot T_{TRACE}(\mathcal{F})\}$ and of size $O(|A_R|)$.

Moreover, let $\tau_\diamondsuit$ be the linear bottom-up tree transducer on $T_{TRACE}(\mathcal{F} \cup \{\diamondsuit\})$ realizing the projection on $\Sigma \cup \Gamma$, i.e. removing the diamond symbol, and $\tau_{\diamondsuit^{-1}}$ be the bottom-up tree transducer on $T_{TRACE}(\mathcal{F} \cup \{\diamondsuit\})$ inserting random diamonds on the ouput. $\tau_\diamondsuit$ and $\tau_{\diamondsuit^{-1}}$ are of constant size.

Then $R$ is realized by the linear bottom-up tree transducer $\tau_\diamondsuit \circ \tau_{C'''} \circ \tau_T$. Indeed, for all $t_1, t_2 \in T_{TRACE}(\mathcal{F}) \,, \alpha \in T_{ACTION}(\mathcal{F}_\Gamma)$, let $t' = t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2$, then:

$$\begin{gathered}
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R \\
\Leftrightarrow \\
t' = t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \in Img_\diamondsuit(R) \\
\Leftrightarrow \\
\text{by (2)} \\
t' \in T_{TRACE}(\mathcal{F}) \cdot C''' \cdot T_{TRACE}(\mathcal{F}) \\
\Leftrightarrow \\
(t_1 \cdot t_2, t') \in T, (t', t') \in R_{\tau_{C'''}} \text{ and } (t', t_1 \cdot \alpha \cdot t_2) \in R_{\tau_\diamondsuit} \\
\Leftrightarrow \\
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R_{\tau_\diamondsuit \circ \tau_{C'''} \circ \tau_T} \,.
\end{gathered}$$

Similarly, $R^{-1}$ is realized by the linear bottom-up tree transducer $\tau_{T^{-1}} \circ \tau_{C'''} \circ \tau_{\diamondsuit^{-1}}$. Indeed: $\left(R_{\tau_\diamondsuit}\right)^{-1} = R_{\tau_{\diamondsuit^{-1}}}$ and $\left(R_{\tau_{C'''}}\right)^{-1} = R_{\tau_{C'''}}$, hence:

$$R^{-1} = R_{\tau_{T^{-1}} \circ \tau_{C'''} \circ \tau_{\diamondsuit^{-1}}} \,.$$

**Theorem 4.** *Let $R$ be a terminating 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 $(m,n)$-completeness for some positive integers $m$ and $n$.*

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

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

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 $\left(R^{-1}\right)^{\leq n}(T_{TRACE}(\mathcal{F} \setminus M'))$ represents the set of unreliably infected traces to avoid. Let's denote by $M''$ its complement: $M'' = T_{TRACE}(\mathcal{F}) \setminus \left(R^{-1}\right)^{\leq n}(T_{TRACE}(\mathcal{F}) \setminus M')$. So:

$$R^{\leq n}(t') \subseteq M' \Leftrightarrow t' \in M''.$$

The property of $(m, n)$-completeness can be restated as follows:

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

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

We use the following lemmas to prove Theorem 5.

**Lemma 3.** *Let $\tau$ be a linear bottom-up tree transducer and let $A$ be a linear bottom-up tree automaton. $\tau(\mathcal{L}(A))$ is a regular tree language recognized by a linear bottom-up tree automaton of size $O(|\tau| \times |A|)$.*

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

**Lemma 4.** *Let $R$ be a terminating abstraction relation. Let $M$ be a regular malicious behavior with the property of $(m,n)$-completeness for some positive integers $m$ and $n$.*

*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 3, as is their union for $1 \leq i \leq n$ the complement of their union.

**Theorem 5.** *Let $R$ be a terminating abstraction relation such that $R$ and $R^{-1}$ are rational. Let $\tau$ be a linear bottom-up tree transducer realizing $R$. Let $M$ be a regular malicious behavior with the property of $(m,n)$-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 $O\left(|\tau|^{m\cdot(m+1)/2} \times |A| \times |A_M|\right)$ 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:

$$L \models M \Leftrightarrow R^{\leq m}(L) \cap M'' = \emptyset$$

By Lemma 3, there is a tree automaton recognizing $R^{\leq m}(L)$ of size $O\left(|\tau|^{m\cdot(m+1)/2} \times |A|\right)$. 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.