Behavior Abstraction in Malware Analysis - Extended Version

Philippe Beaucamps, Isabelle Gnaedig, Jean-Yves Marion

Behavior Abstraction in Malware 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 present an approach for proactive malware detection working by abstraction of program behaviors. Our technique consists in abstracting program traces, by rewriting given subtraces into abstract symbols representing their functionality. Traces are captured dynamically by code instrumentation, which allows us to handle packed or self-modifying malware. Suspicious behaviors are detected by comparing trace abstractions to reference malicious behaviors. The expressive power of abstraction allows us to handle general suspicious behaviors rather than specific malware code and then, to detect malware mutations. We present and discuss an implementation validating our approach.

Keywords: Malware, behavioral detection, behavior abstraction, trace, string rewriting, finite state automaton, formal language, dynamic binary instrumentation.

1 Introduction

Detection techniques of computer malware have traditionally relied on a combination of static and dynamic analysis. A shortcoming of static analysis, however, is the general intractability of knowing in advance the entire program code as it may change dynamically. Packing and obfuscation techniques typically capitalize on this intractability to prevent the reconstruction of the program code. Structural and behavioral techniques, on the other hand, may be used to guard against code protection and code transformation and are consequently more robust and reliable. These techniques rely on the analysis of the program structure or its behavior rather than its binary code. This ensures an independence from basic syntactic modifications or from packing techniques.

Structural techniques analyze the control flow graph representing the program, by assuming that its structure remains untouched. They compare it to control flow graphs of known malware using static analysis [10], sub-graph isomorphism [9], tree automata [6] or similarity measures [12,4]. Unfortunately, these techniques are not resilient to functional polymorphism of behaviors, of which malware variants are a form, and which expresses that behaviors can

* This work has been partially supported by the High Security Lab of the LORIA in Nancy: http://lhs.loria.fr.

be carried out in different ways, without their functionality being altered. This polymorphism often impacts the structure of the control flow graph.

Conversely, behavioral approaches, first introduced in Cohen's seminal work [11], monitor system calls and their arguments and have traditionally relied on the use of finite state machines [18,21]. Recent approaches [19] deal with functional polymorphism 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. Subsequently, [16] proposed to use attribute automata but the cost is an exponential time complexity procedure.

Other behavioral approaches also use model checking techniques to track data [5,17,22]: they define behavioral signatures as temporal logic formulae, defined on a syntactic level. But none of these approaches considers functional polymorphism. Moreover they do not tackle either the problem of constructing a high-level view of a program, which limits their applicability.

Our goal here is to provide a generic framework for malware detection, abstract enough to be independent of the implementation of programs, resilient to variants and relying on general suspicious behaviors rather than on specific malware code. Unlike the approaches cited before, which only consider the detection scenario by working on one trace at a time, we intend to make our formalism more generally applicable to analysis and signature generation of unkown malware by working on a set of traces representing the whole behavior of a program.

For this purpose, we present an approach working on an abstract representation of the behavior of a program. We propose an original strong theoretical setting underpinned by the theory of formal languages and based on string rewriting systems and finite state automata. Abstraction is carried out with respect to behavior patterns defined by string rewriting systems. Behavioral detection is then carried out by automata intersection.

More precisely, execution traces of a program describe the capture of specific data such as program instructions, system calls with their arguments, or file system and network interactions. We represent a set of execution traces by an automaton called trace automaton. An abstraction of this trace automaton is then constructed, with respect to a set of predefined behavior patterns defined as regular languages and describing high-level properties or actions such as a file operation, a hook installation or a data leak. This gives a representation independent of the program implementation. Finally, the abstracted trace automaton is intersected with a malware database, composed of abstract signatures representing known malicious behaviors.

Our technique offers two detection scenarios: identifying a program as suspicious when it performs some malicious action like keylogging, or detecting an action sequence similar to the one of a known malware. The model we use, combining string rewriting systems and finite state automata, allows us to detect very efficiently malicious behaviors with high level descriptions, that is in linear time. Detection speed can be tuned by setting the right tier of abstraction level. So our behavioral detection model could be used inside a firewall for example.

After presenting the background in Section 2, we define behavior patterns in Section 3. Section 4 presents the abstraction mechanism of trace languages. Section 5 explains how to represent trace languages by trace automata and gives complexity bounds for computing abstractions. Section 6 formalizes the detection problem with its cost. Section 7 presents the implementation of our approach together with experiments. We conclude in Section 8.

2 Background

Let Σ be some finite alphabet. We denote by Σ^* the set of finite words on Σ. Subsets of Σ^* are called languages on Σ. The empty word is denoted by ϵ.

Let Σ' be some finite alphabet. The projection homomorphism which maps words of Σ^* to words of Σ'^* is denoted by $u|_{\Sigma'}$ for any $u \in \Sigma^*$ and is defined, for $a \in \Sigma$, by: $a|_{\Sigma'} = a$ if $a \in \Sigma'$ and $a|_{\Sigma'} = \epsilon$ otherwise. This definition is homeomorphically extended to languages on Σ, and the projection on Σ' of a language L on Σ is denoted by $L|_{\Sigma'}$.

A finite state automaton $\mathcal{A}$ on an alphabet Σ is a tuple (Q, δ, q_0, F) where Q is a finite set of states, $\delta : Q \times \Sigma \to Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A run of $\mathcal{A}$ on a word $w = a_0 a_1 \cdots a_n$ is a sequence of states $r = q_0 q_1 \cdots q_{m \leq n+1}$ such that: $\forall i < m, q_{i+1} \in \delta(q_i, a_i)$. The run r is successful if $m = n + 1$ and $q_m \in F$; in this case, w is said to be recognized by $\mathcal{A}$. The set of words for which there exists a successful run of $\mathcal{A}$ is the language recognized by $\mathcal{A}$ and is denoted by $\mathcal{L}(\mathcal{A})$. Languages recognized by some finite state automaton $\mathcal{A}$ are called regular. The size of a finite state automaton A, denoted by $|A|$, is defined as the number of states of A.

For a given binary relation $\to$, we denote by $\to^*$ its reflexive transitive closure. A string rewriting system (SRS in short) is a triple $(\Sigma, V, \to)$, where V is a set of variables and $\to$ a binary relation on $(\Sigma \cup V)^*$. A regular SRS is a 4-tuple $(\Sigma, V, S, \to)$, where $S \in V$ and the relation $\to$ is generated by rules of the form $a \to A$, $aA \to B$ and $\epsilon \to A$, with $a \in \Sigma$, $A, B \in V$. The language recognized by a regular SRS is the set $\{u | u \in \Sigma^*, u \to^* S\}$. Languages recognized by regular SRS are exactly the regular languages. The size of a regular SRS R, denoted by $|R|$, is defined as the number of variables of V.

3 Behaviors

We now introduce a model of abstract machine from which we define notions of execution trace and behavior. An abstract machine $\mathcal{M}$ consists of the triple $(\mu_0, IP_0, \to)$ where (μ_0, IP_0) is an initial configuration of $\mathcal{M}$ and $\to$ is a transition function from *Configurations* to *Configurations*, where *Configurations* denotes the set of configurations of $\mathcal{M}$.

A configuration of $\mathcal{M}$ is a pair (μ, IP) where:

- $\mu : \mathit{Addresses} \to \mathit{Data}$ represents the memory of $\mathcal{M}$. *Addresses* is the set of addresses of $\mathcal{M}$ and *Data* is a set of values; both are subsets of $\mathbb{N}$;

– $IP \in Addresses$ is the instruction pointer.

Thus, we have $(\mu, IP) \to (\mu', IP')$ if the machine $\mathcal{M}$ executes the instruction at address IP of memory μ. The memory μ' is the memory obtained after executing this instruction and IP' is the address of the next instruction to execute. A program is a set of instructions. An *execution* of an abstract machine $\mathcal{M}$ is a finite sequence:

$$(\mu_0, IP_0) \to (\mu_1, IP_1) \to \ldots \to (\mu_n, IP_n)\,.$$

In our scenario, the configuration (μ_0, IP_0) is the initial configuration. A program is loaded into μ_0 at the address pointed by the instruction pointer IP_0. So, at the beginning of an execution, a program is executed inside an initial environment (also) given by μ_0. Then, the program is run. At each step, we see interactions with the "outside" through the memory. Though our model of abstract machine can only represent single-threaded programs, this formalization of communications is enough for our purpose.

The reader will notice that we focus on abstract machines rather than on programming languages. There are several reasons for this. First of all, our model allows us to talk about programming languages at any level of abstraction. Second, in the context of malware, programs are generally self-modifying. Programs are treated as data objects and elements of *Data* are regarded as instructions. A program text is variable, which is not the usual point of view in semantics, see for example Gunter's textbook [15]. Moreover, low level instructions, like in the x86 case, are not of fixed size and a program can modifiy its code by instruction misalignment. So we think that our model of abstract machine is a right approach to underpin our study on malware behavior detection.

As said before, dynamic analysis and detection rely on capture and analysis of execution data. This data may be the sequence of instructions being executed, the sequence of system calls (i.e. calls to system code), etc. Other approaches may capture yet higher-level actions of a program, for instance network interactions, file system accesses, inter process communications (IPC), register usage statistics or any data that can be used to characterize a behavior. Our framework aims at dealing with any kind of the above data using an alphabet Σ.

We first formalize the capture of some execution data, represented by elements of Σ, in the machine $\mathcal{M}$.

Definition 1 (Capture operator). *A* capture operator *with respect to Σ is an operator $\pi : Configurations \to \Sigma \cup \{\epsilon\}$ which associates with some configuration the captured data if any, and ϵ otherwise.*

Note that in the general case, captured data of a configuration c_n may depend on the history of the execution i.e. on configurations $c_{i_1} \ldots c_{i_k}$ for $i_1 \ldots i_k \in [1..n-1]$ at previous times. This is needed for example to compute register usage statistics or to capture non atomic behaviors (for instance "smtp connection" which is the combination of a network connection and reception of the message "220 .* SMTP Ready"). For the sake of simplicity, for defining π, we do not consider $c_{i_1} \ldots c_{i_k}$.

From now on, in our examples, we consider more specifically the operator capturing library calls, including system calls.

Definition 2 (Execution trace). *Let $\mathcal{M}$ be a machine, $e = c_1 \dots c_n$ an execution of $\mathcal{M}$ and π a capture operator with respect to Σ. Then $\pi(c_1) \dots \pi(c_n) \in \Sigma^*$ is the* execution trace *of the execution e of $\mathcal{M}$ with respect to π, denoted $\pi(e)$.*

In the following, we will call trace language of a machine $\mathcal{M}$ with respect to the capture operator π, the set of execution traces of $\mathcal{M}$ with respect to π. We denote it by $Traces_\pi(\mathcal{M})$, or simply $Traces(\mathcal{M})$ when π is clear from the context. We can now formally define the notion of behavior we want to detect in a program, with respect to some capture operator.

Definition 3 (Behavior pattern). *A* regular behavior pattern $\mathcal{B}$ *is a regular language on Σ. We call* simple behavior pattern *an element of $\mathcal{B}$.*

A behavior pattern does not necessarily describe a malicious behavior in itself. It can describe an innocuous behavior – e.g. creating a process, opening a file, sending a mail – or a relevant behavior sequence, possibly modulo shuffling of independent actions. It can also represent a more specific behavior which is shared by different malicious codes.

Example 1. Throughout this paper, we consider the example of the Allaple worm, a polymorphic network worm. A simplified excerpt of the code of its variant Allaple.A is given in Appendix B. It contains three behavior patterns: the ping of a remote host, the opening of a Netbios connection and the scanning of local drives. An example of execution trace of this excerpt is the following sequence of library calls:
`...GetLogicalDriveStrings.GetDriveType.FindFirstFile.FindFirstFile.FindNextFile...`

This trace exhibits the behavior pattern which describes the scanning of local drives: `GetLogicalDriveStrings.GetDriveType.FindFirstFile`.

4 Trace Abstraction

Given some machine, recall that our goal is to abstract its execution traces, by replacing concrete behavior patterns by more abstract representations, expressing their functionality, in order to compare them to reference malicious behaviors.

In this section, we formally define the abstraction of a program trace with respect to behavior patterns. We start with a simple behavior pattern, then generalize abstraction to a regular behavior pattern and finally to a set of regular behavior patterns. We show that abstraction can be viewed as a string rewriting process where each behavior pattern is replaced by a new specific symbol.

4.1 Abstracting with respect to a Simple Behavior Pattern

Let $t \in \mathcal{B}$ be a simple behavior pattern for some $\mathcal{B}$. Let $\mathcal{M}$ be a machine and $u \in Traces_\pi(\mathcal{M})$ an execution trace of $\mathcal{M}$ for some π. Identifying occurrences of the pattern t in the trace u amounts to matching t to some subword of u.

Let $\lambda \notin \Sigma$ be a new symbol denoting the abstraction of our pattern t. We define $\Sigma' = \Sigma \cup \{\lambda\}$. Trace abstraction of u with respect to pattern t of Σ^* is defined by rewriting u with the string rewriting system R on Σ'^*, where R is composed of a single rewrite rule: $R = \{t \to \lambda\}$.

Let $\to_t$ denote the rewriting relation induced by R, which rewrites substrings:

$$\forall u, v \in \Sigma'^*, u \text{ rewrites into } v, \text{ which is written } u \to_t v \text{ iff}$$
$$\exists u' \in \Sigma'^*, \exists u'' \in \Sigma'^*, \begin{cases} u = u' \cdot t \cdot u'' \\ v = u' \cdot \lambda \cdot u'' \,. \end{cases}$$

A trace has been abstracted when every occurrence of the pattern t has been replaced by the abstract symbol λ. Thus abstraction of a trace is nothing but normalization of that trace with respect to the SRS R. Note that R is not confluent in general, so a trace may have several normal forms. Abstraction of a trace can be naturally generalized to a trace language.

Example 2. Returning to our excerpt of the Allaple worm, suppose we are interested in detecting the previously defined behavior of scanning local drives. Then $t =$ `GetLogicalDriveStrings.GetDriveType.FindFirstFile`, and we define $\lambda =$ `SCAN_DRIVES`. The execution trace in Example 1 is thus abstracted into:
`...SCAN_DRIVES.FindFirstFile.FindNextFile...`

Note that we consider normal forms instead of partially reduced ones. In other words, we require that every occurrence of a pattern that can be matched is eventually rewritten. This allows to ensure that the computation of the malicious behavior exhibited by a given malware is maximal. Also, this allows a more precise detection of behaviors of the type "A followed by B, without C in between": partially rewriting a trace acb into AcB would then lead to a positive detection, while the normal form ACB does not match the behavior.

4.2 Abstracting with respect to a Regular Behavior Pattern

A behavior pattern can usually be achieved in different ways and malware writers may capitalize on this to perform functional polymorphism. For instance, creating a file may be processed using different sequences of instructions: detection will only be interested in the functionality expressed by these sequences, not in their implementation details. Thus, we now consider the case of a behavior pattern defined as a language, and more specifically as a regular language, as presented in Definition 3. Indeed, regular patterns allow to account for example for behaviors interleaved with other unrelated or irrelevant actions. For instance, file writes may often be interleaved with other interactions with the system, e.g. string manipulation functions.

Normalization now consists in rewriting a trace language with the rules transforming into λ any simple pattern of a behavior pattern $\mathcal{B}$, where $\mathcal{B}$ is a regular language on Σ^*. Since $\mathcal{B}$ may be infinite, we consider a regular SRS $(\Sigma, V, S, \to)$ recognizing $\mathcal{B}$ and we define relation $\to_{\mathcal{B}}$ on $(\Sigma' \cup V)^*$ as the rewriting relation induced by $\to \cup \{S \to \lambda\}$. Using the reflexive transitive closure of $\to_{\mathcal{B}}$, we define trace language abstraction with respect to a regular behavior pattern.

Definition 4 (Abstract trace language). *Let $\mathcal{B}$ be a regular behavior pattern, and $\rightarrow_{\mathcal{B}}$ its associated rewriting relation. The* abstract form *of the trace language L with respect to $\mathcal{B}$, denoted by $L\downarrow_{\mathcal{B}}$, is defined by:*

$$L\downarrow_{\mathcal{B}} = \{v \in \Sigma'^* \mid \exists u \in L,\ u \rightarrow_{\mathcal{B}}^* v \text{ and } \nexists w \in \Sigma'^*,\ v \rightarrow_{\mathcal{B}} w\}\ .$$

The following theorem expresses that abstraction preserves regularity, which is a fundamental characteristic of our approach.

Theorem 1. *Let $\mathcal{B}$ be a regular behavior pattern and L a trace language. If L is regular, then so is $L\downarrow_{\mathcal{B}}$.*

Proofs of all theorems of the paper are given in Appendix A.

Example 3. One could extend the previous pattern `SCAN_DRIVES` to account for alternative library calls, e.g. calling `FindFirstFileEx` instead of `FindFirstFile`: `SCAN_DRIVES = GetLogicalDriveStrings.GetDriveType.(FindFirstFile +FindFirstFileEx)`. This corresponds to the following SRS:

$$\begin{gathered}\texttt{FindFirstFile} \rightarrow A\\ \texttt{FindFirstFileEx} \rightarrow A\\ \texttt{GetDriveType}.A \rightarrow B\\ \texttt{GetLogicalDriveStrings}.B \rightarrow \texttt{SCAN\_DRIVES}\,.\end{gathered}$$

4.3 Abstracting with respect to a Set of Regular Behavior Patterns

Finally, we generalize abstraction to a set of behavior patterns. Indeed, in practice, a suspicious behavior can be expressed as the combination of several behavior patterns $\mathcal{B}_i$, each of them abstracting into a distinct symbol λ_i.

Throughout this paper, we denote by Γ the set of symbols representing abstractions of our behavior patterns and we extend Σ' to $\Sigma \cup \Gamma$.

Let $\mathcal{C} = \{\mathcal{B}_i \mid \mathcal{B}_i \subseteq \Sigma^*\}_{1\leq i\leq n}$ be a finite set of regular behavior patterns respectively associated with distinct symbols $\lambda_i \in \Gamma$.

As a relation $\rightarrow_{\mathcal{B}_i}$ is defined on $(\Sigma' \cup V_i)^*$, the relation $\rightarrow_{\mathcal{C}} = \bigcup\limits_{1\leq i\leq n} \rightarrow_{\mathcal{B}_i}$ is defined on $(\Sigma' \cup \bigcup\limits_{1\leq i\leq n} V_i)^*$. We extend trace language normalization to a set of regular behavior patterns, and the trace language L is now normalized with $\rightarrow_{\mathcal{C}}$ into the abstract trace language $L\downarrow_{\mathcal{C}}$.

Theorem 2. *Let $\mathcal{C}$ be a finite set of regular behavior patterns. If L is regular, then so is $L\downarrow_{\mathcal{C}}$.*

4.4 Projecting the Abstract Trace Language on Γ

Once a trace language has been normalized with respect to some set of behavior patterns, details that have not been processed by the normalization have to be pruned. A completely abstracted trace language is indeed expected to be defined on Γ, in order to be compared afterwards to reference abstract behaviors. This pruning operation is performed by projecting the abstract trace language on Γ.

Definition 5 (Γ-abstract trace language). *Let $\mathcal{C}$ be a set of regular behavior patterns. Let L be a trace language for some machine $\mathcal{M}$. $\widehat{L} = L\downarrow_{\mathcal{C}}|_{\Gamma}$ is called the* Γ-abstract trace language *of $\mathcal{M}$ with respect to $\mathcal{C}$.*

Once the abstraction is complete, the simplified traces describe functionality-related actions and are consequently more robust and less complex. As they represent several different implementations of the same behavior, they allow to deal with functional polymorphism.

We can then compare the language of Γ-abstracted traces to the behavior of some known malware or to some generic malicious behavior defined on Γ.

Finally, the whole abstraction process could be repeated, as in Martignoni et al's layered architecture [19]. A first layer would look up behavior patterns defined in terms of raw analysis data. A second layer would look up behavior patterns defined in terms of patterns from the first layer, and so on. However, this case is encompassed in our formalism. It suffices to define behavior patterns from the final layer directly in terms of the raw analysis data, by composition of the regular SRSs defining patterns of the different layers. The resulting patterns thereby remain regular on Σ, so our formalism can still be applied.

5 Trace Abstraction Using Finite State Automata

When considering a single trace, the associated trace language is trivially represented by an automaton. But when this trace language describes the set of traces $Traces(\mathcal{M})$ of some machine $\mathcal{M}$, this set is in general undecidable or at least non-regular, so no automaton can precisely represent it. Nevertheless, one may build a regular approximation of this trace language, which is usually twofold:

- the trace language is over-approximated, when replacing for instance $a^n \cdot b^n$ sequences (stemming from program loops for instance) by $a^* \cdot b^*$ sequences or when coping with obfuscation or dynamic analysis shortcomings. The resulting automaton then contains spurious traces (false positives).
  Formally, if $Traces(\mathcal{M})$ is over-approximated by L, then $Traces(\mathcal{M}) \subseteq L$.
- the trace language is under-approximated, when some hidden code path is not discovered or when some uninteresting path is discarded. The resulting automaton then misses some traces (false negatives).
  Formally, if $Traces(\mathcal{M})$ is under-approximated by L, then $L \subseteq Traces(\mathcal{M})$.

Thus a trace automaton represents a regular approximation of the trace language of some machine and can be defined as an automaton recognizing a part of the traces of $\mathcal{M}$.

Definition 6 (Trace automaton). *Given a machine $\mathcal{M}$, a* trace automaton *for $\mathcal{M}$ with respect to Σ is a finite-state automaton A on Σ such that:*

$$\exists S \subseteq \Sigma^*, S \subseteq Traces(\mathcal{M}) \wedge S \subseteq \mathcal{L}(A) .$$

In order to construct the trace automaton of a machine, one may either use a collection of captured traces or reconstruct the program machine code from its binary representation. In the first case, the automaton is built in such a way that the captured traces correspond to a path in the trace automaton. In the second case, the machine code of the program can be reconstructed using common techniques that combine static and dynamic analysis: it is then projected on the trace alphabet and the trace automaton is inferred from the code structure.

Example 4. Figure 1 shows a trace automaton for the Allaple.A excerpt representing the ping of a remote host and the scanning of local drives.

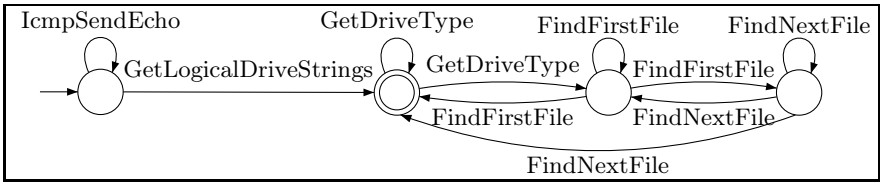

Fig. 1. Trace automaton for the Allaple.A excerpt

By Theorem 2, the abstraction problem for a regular trace language now amounts to computing an automaton recognizing the abstraction of this language. Construction of this automaton, which we call abstract trace automaton, is described in the proofs of the following theorems and uses a method proposed by Esparza et al. [13]. It consists in modifying the initial trace automaton by adding new transitions using the left hand sides of the rewriting rules and then intersecting it with an automaton recognizing the words in normal form with respect to our rewrite system.

Thus, the abstract trace automaton may be more complex than the initial one, as shown by the Allaple worm example. Abstraction of the trace automaton with respect to patterns `SCAN_DRIVES` and `PING`, where `PING` = `IcmpSendEcho` describes the ping of a remote host, gives the automaton of Figure 2.

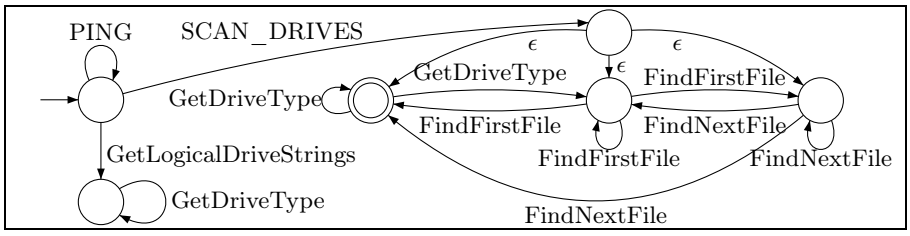

Fig. 2. Abstract trace automaton for the Allaple.A excerpt

Theorem 3. *Let A be a trace automaton and $\mathcal{C} = \{\mathcal{B}_i\}_{1 \leq i \leq n}$ a set of behavior patterns recognized by regular SRSs $\{R_{\mathcal{B}_i}\}_{1 \leq i \leq n}$. Let $|\mathcal{C}| = \sum_{1 \leq i \leq n} |R_{\mathcal{B}_i}|$.*

Then an automaton of size $O(|A|)$ recognizing $\mathcal{L}(A) \downarrow_{\mathcal{C}} |_{\Gamma}$ can be constructed in time $O\left(|A|^3 \cdot |\mathcal{C}|^2\right)$ and space $O\left(|A|^2 \cdot |\mathcal{C}|^2\right)$.

The final abstraction of the Allaple.A excerpt, for $\Gamma = \{\texttt{PING}, \texttt{SCAN\_DRIVES}\}$, is depicted in Figure 3.

Fig. 3. Γ-abstract automaton for the Allaple.A excerpt

6 Application to Malware Detection

Using the abstraction framework defined in Section 4, malware detection now consists in computing the abstract trace language of some machine and comparing it to a database of malicious behaviors defined on Γ. These malicious behaviors either describe generic behaviors, e.g. sending spam or logging keystrokes, or behaviors of specific malware. According to our abstraction formalism, malicious behaviors are sets of particular combinations of behavior patterns abstractions.

Definition 7. *A* malicious behavior *on Γ, or* signature*, is a language on Γ.*

More specifically, a malicious behavior describes combinations of patterns, possibly interleaved with additional patterns which are irrelevant in these combinations. For instance, we define the signature for the Allaple worm as the following regular language, which explicitly allows interleaving of patterns that do not match the upcoming pattern:

$$\texttt{LOCAL\_COM\_SERVER} \cdot (\Gamma \setminus \{\texttt{PING}\})^* \cdot \texttt{PING} \cdot (\Gamma \setminus \{\texttt{NETBIOS\_CONNECTION}\})^* \cdot \texttt{NETBIOS\_CONNECTION}.$$

The automaton representing the signature of Allaple is given in Figure 4. Note that the `SCAN_DRIVES` pattern, which is present in the Γ-abstract trace automaton of the Allaple.A excerpt, does not appear here because the signature describes a common discriminating behavior exhibited by all samples of Allaple.

Definition 8. *Let L_m be a malicious behavior on Γ. A machine $\mathcal{M}$, with a Γ-abstract trace language $\widehat{L}$,* exhibits the malicious behavior L_m *iff there exists $v \in L_m$ and $u \in \widehat{L}$ such that $u = u_1 v u_2$, where $u_1, u_2 \in \Gamma^*$.*

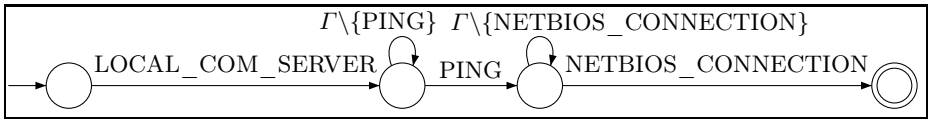

Fig. 4. Allaple signature

Thus, $\mathcal{M}$ exhibits the behavior L_m if some subword of an abstract trace of $\widehat{L}$ is in L_m. Our *malicious database* is then a set $\mathcal{D}$ of malicious behaviors. A machine $\mathcal{M}$ is *malicious* with respect to $\mathcal{D}$ if it exhibits some malicious behavior of $\mathcal{D}$.

$\widehat{L}$ can be constructed either from a single captured trace or from a whole trace language. In the first case, the detection process is lighter. In the second case, we get a better description of the program behavior, so detection is more reliable.

When $\widehat{L}$ is represented by an automaton A and L_m by an automaton A_m, $\mathcal{M}$ exhibits the behavior L_m when the following holds. Let A'_m be the automaton recognizing the set of words containing a subword in L_m: A'_m is constructed from A_m by adding loops labelled by symbols of Γ on the initial state and the final states of A_m. Then $\mathcal{M}$ exhibits the behavior L_m iff $\mathcal{L}(A) \cap \mathcal{L}(A'_m) \neq \emptyset$.

The malicious database is then represented by an automaton $A_{\mathcal{D}}$ which is the union of the automata A_m representing the signatures and may be minimized.

Theorem 4. *Let $\mathcal{D}$ be a set of regular malicious behaviors on Γ, recognized by an automaton $A_{\mathcal{D}}$. Let $\mathcal{M}$ be a machine, with a Γ-abstract trace language recognized by an automaton A. Then deciding whether $\mathcal{M}$ is malicious with respect to $\mathcal{D}$ takes time: $O\left(|A_{\mathcal{D}}|^2 \cdot |A|^2\right)$.*

Note that runtime detection could be efficiently implemented from these results, since all constructions (Γ-abstraction, automata intersection) are incremental.

Now, infection could be defined more intuitively and more generally in the following way: behavior patterns, instead of representing building blocks of a malicious behavior, could directly represent a malicious behavior, which would then be defined on Σ.

Definition 9. *Let $\mathcal{C}$ be a set of behavior patterns. Let $\mathcal{M}$ be a machine, with a trace language L. $\mathcal{M}$ is* malicious *if $L\downarrow_{\mathcal{C}}|_{\Gamma} \neq \{\epsilon\}$.*

With the above detection model, we lose the expressive power of abstraction and the robustness of our detection model. But on the other hand, detection is performed by the sole process of normalization.

7 Implementation and Experiments

We implemented the presented techniques in a tool which is able to capture execution traces of a given program, to build its trace automaton, to abstract it with respect to a set of predefined behavior patterns and to compare it to a malware database.

Setting up the detection environment In order to avoid static analysis shortcomings and to ignore unreachable code, we use dynamic analysis to construct a trace automaton for some program loaded into a machine. The program is instrumented using Pin [3], which allows us to collect library calls along with their arguments while the program is running. Other instrumentation tools, like Dynamorio [1], could have been utilized with similar results. Instrumentation allows us to perform low-level analysis of the execution and to have a tight control over the program. In particular, our instrumentation tool handles threads, child processes and hooks, and features a simple data flow analyzer which relates call arguments to previously used data. In order to reduce the size of captured traces and to capture the behavior of a program at a source code level, we only collect library calls directly made from the program code, ignoring calls originating from libraries themselves.

When an execution trace is captured, we construct a trace automaton by associating a state with each different instruction pointer responsible for making a library call. Threads are handled by associating a state with a set of instruction pointers. Additional execution traces can be used to complete the trace automaton. Automata are manipulated with the OpenFST library [2].

Behavior patterns are defined after observing malicious execution traces and extracting basic sequences likely to be part of a malicious behavior. These patterns often define usual interactions of the program with the system or the network. Once extracted, a sequence either defines a new behavior pattern or extends an existing pattern.

The malware database is a collection of malicious behaviors, which is built from a set of generic signatures along with signatures of known malware, constructed using their Γ-abstract trace automata. The resulting database automaton is minimized in order to speed up detection.

Experiments To test our detection method, samples of malicious programs were collected using a honeypot[1] and identified using Kaspersky Antivirus.

We defined 40 behavior patterns, extended to allow data constraints expressing properties about the arguments of the calls. These constraints are compatible with our formalism and amount to modify the trace automaton by adding transitions denoting the verification of a constraint (See Appendix C.1). Examples of such patterns include writing to system files, persisting oneself in the Windows registry or browsing drives and files (see Appendix C.2).

Three experimentation scenarios were defined. In a first setting, we define signatures for given malware families, by analysis and Γ-abstraction of a few samples. Samples from different variants of each family are then compared to the signatures: several of these variants are new, i.e. they were not considered when defining the signatures. This allows us to test the applicability of our approach and its robustness against mutation. In a second setting, a more general signature is defined for a common malicious behavior encountered in different malware families. Several malicious samples are then tested, in order to find out

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

which samples exhibit this behavior. This allows us to test the expressive power of behavior abstraction. In a third setting, sane applications are tested, to ensure that the rate of false positives is low.

We tested the above settings on known malware families, among which Allaple, Virut, Agent, Rbot, Afcore and Mimail. In particular, our honeypot shows that Allaple, Virut and Agent are currently among the most active worms, which makes their analysis particularly relevant. Most samples were successfully matched to the defined signatures, while failures stemmed from technical shortcomings of our implementation or of dynamic analysis. Experiments results of the previous scenarii are given in Appendix C, along with a complete abstraction example for the Agent.ah worm.

8 Conclusion and Future Work

In this paper, we have presented a new malware detection approach using abstract representations of malicious behaviors to identify malicious programs. Programs to analyze, represented as trace languages, are abstracted by rewriting with respect to elementary behavior patterns, defined as regular string rewriting systems. Abstractions are then compared to a database of abstract malicious behaviors, wich describe combinations of the former patterns.

Abstraction is the key notion of our approach. Providing an abstracted form of program behaviors and signatures allows us to be independent of the program implementation, to handle similar behaviors in a generic way and thus to be robust with respect to existing and future variants. The strength of our technique lies also in the fact that abstract malicious behaviors are combinations of elementary patterns: this allows us to efficiently summarize and compact the possible combinations likely to compose suspicious behaviors. Moreover, malicious behaviors are easy to update since they are expressed in terms of basic blocks. Behavior patterns themselves, as they describe basic functionalities, are easier to update than if they had described more complex ones.

Behavior abstraction may also prove useful in similarity analysis of malicious programs. Like for detection, the use of abstract representations of programs makes the analysis resilient to functional polymorphism and to minor changes.

We plan to extend our approach in several directions. The first one is concerned with data flow analysis. The behavior patterns we define are unable to express that the actions they match are actually related. Data flow information would allow to address this question. While this issue is not very relevant when matching local behaviors (eg. writing data to a file), it becomes more important when specifying wider behaviors which spread over large execution sequences. Work is in progress to handle data flow by using model checking with temporal logic formulae with parameters.

The second extension is concerned with interleaved behavior pattens. When two behavior patterns are interleaved in an execution trace, our approach can only match one pattern out of the two since, while rewriting the first one, the second will be consumed. Although interleaving does not occur very often in

practice since common behavior patterns are small and not very prone to interleaving, we intend to propose non-consuming approaches to pattern abstraction.

Also, we would like to improve the construction of a trace automaton approximating a trace language. When it is built from execution traces, meaningless paths are created by interference between relevant paths. This increases the matching possibilities between the abstract trace automaton and the malicious signatures, which impacts the precision of the detection. A solution would be to duplicate function calls in the automaton when appropriate.

Lastly, captured traces do not give an exhaustive view of all possible executions and some interesting behaviors may only be observed when some conditions are met. We could use existing tools identifying these conditions. Sage [14] and BitScope [8] use symbolic execution to systematically discover new execution paths. Moser et al. [20] also address this problem in the malware context by monitoring how certain inputs impact the control flow and by instrumenting these inputs to drive the execution.

References

1. DynamoRIO. `http://dynamorio.org`.
2. OpenFST. `http://www.openfst.org/`.
3. Pin. `http://www.pintool.org`.
4. Martin Apel, Christian Bockermann, and Michael Meier. Measuring similarity of malware behavior. In *IEEE Conference on Local Computer Networks*, pages 891–898. IEEE, October 2009.
5. 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.
6. Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 2008.
7. Ronald V. Book and Friedrich Otto. *String-rewriting systems*. Springer-Verlag, London, UK, 1993.
8. David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. *Botnet Detection*, 36:65–88, 2008.
9. Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2006.
10. Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46. IEEE Computer Society, 2005.
11. Fred Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6(1):22–35, 1987.
12. Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Télécommunications*, 2005.
13. Javier Esparza, Peter Rossmanith, and Stefan Schwoon. A uniform framework for problems on context-free grammars. *Bulletin of the EATCS*, 72:169–177, 2000.

14. Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*. Internet Society, 2008.
15. Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
16. 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.
17. 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.
18. 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.
19. 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.
20. Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245. IEEE Computer Society, 2007.
21. 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.
22. 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.
23. Ferucio Laurentiu Tiplea and Erkki Mäkinen. On the complexity of a problem on monadic string rewriting systems. *Journal of Automata, Languages and Combinatorics*, 7(4):599–609, 2002.

A Proofs

Theorems 1 and 2 follow from Theorems 5 and 3.

Theorem 5. *Let A be a trace automaton and $\mathcal{B}$ be a regular behavior pattern recognized by a regular SRS $R_{\mathcal{B}}$. Then an automaton of size $O(|A|)$ recognizing $\mathcal{L}(A)\downarrow_{\mathcal{B}}$ can be constructed in time $O\left(|A|^3 \cdot |R_{\mathcal{B}}|^2\right)$ and space $O\left(|A|^2 \cdot |R_{\mathcal{B}}|^2\right)$.*

Proof. Let $R_{\mathcal{B}} = (\Sigma, V, \rightarrow, S)$, and define $\Sigma' = \Sigma \cup \{\lambda\}$. Suppose that L is a regular language. We denote by $post^*(L)$ the set of successors of elements of L by $\rightarrow_{\mathcal{B}}$:

$$post^*(L) = \left\{u \in (\Sigma' \cup V)^* \,|\exists v \in L, v \rightarrow_{\mathcal{B}}^* u\right\}.$$

Regularity of $post^*(L)$ when L is regular was proved by Book and Otto in [7]: methods for the construction of an automaton recognizing $post^*(L)$ are proposed in [13,23].

Now, $L\downarrow_{\mathcal{B}}$ is the set of words of $post^*(L)$ which are in normal form. Let $lhs(\rightarrow)$ be the set of left members of the rules of $\rightarrow$. Then, a word $u \in (\Sigma' \cup V)^*$ is in normal form if no subword of u matches a word of $lhs(\rightarrow)$, i.e.: $u \notin (\Sigma' \cup V)^* . lhs(\rightarrow) . (\Sigma' \cup V)^*$.

Thus, in the case where $L = \mathcal{L}(A)$, we get:

$$\mathcal{L}(A)\downarrow_{\mathcal{B}} = post^*(\mathcal{L}(A)) \cap \overline{(\Sigma' \cup V)^* . lhs(\rightarrow) . (\Sigma' \cup V)^*}.$$

Esparza et al. proposed in [13] a method constructing an automaton A' recognizing $post^*(\mathcal{L}(A))$ in time $O\left(|R_{\mathcal{B}}|^2 \cdot |A|^3\right)$ and space $O\left(|R_{\mathcal{B}}|^2 \cdot |A|^2\right)$. A' has the same number of states as A, but is defined on $\Sigma' \cup V$.

On another hand, one may build an automaton recognizing the set $\overline{(\Sigma' \cup V)^* . lhs(\rightarrow) . (\Sigma' \cup V)^*}$ in the following way.

First we need to construct a deterministic automaton recognizing $lhs(\rightarrow)$. Rules of $\rightarrow$ have the form $a \rightarrow B$, $aA \rightarrow B$ or $S \rightarrow \lambda$, with $a \in \Sigma$, $A \in V$, $B \in V$. Therefore we partition $lhs(\rightarrow)$ as $\{S\} \cup \bigcup_{1 \leq i \leq n} a_i V_i$, with $a_i \in \Sigma$, $V_i \subseteq V \cup \{\epsilon\}$, the a_i being mutually distinct. Start with an automaton containing an initial state i, a final state f and the transition $i \underset{S}{\rightarrow} f$. For each partition $a_i V_i$, add a new state s_i and transitions $i \underset{a_i}{\rightarrow} s_i$ and $s_i \underset{A}{\rightarrow} f$, with $A \in V_i$ and $A \neq \epsilon$. If $\epsilon \in V_i$, then define s_i as a final state. There are at most $|\Sigma|$ partitions (one for each a_i) so the constructed automaton has $O(1)$ states and $O(|V|)$ transitions, which takes space $O(|V|)$. Furthermore, there are $O\left(|V|^2\right)$ rules of $\rightarrow$ and each rule must be read in order to partition $lhs(\rightarrow)$, so this takes time $O\left(|V|^2\right)$.

From this automaton, we build a complete, deterministic automaton recognizing $(\Sigma' \cup V)^* . lhs(\rightarrow) . (\Sigma' \cup V)^*$. For each non final state s_i coming from

the partition $a_i V_i$, add transitions $s_i \underset{a}{\rightarrow} i$, for each $a \in (\Sigma' \cup V) \setminus V_i$. Finally, add a $\Sigma' \cup V$ loop on each final state. For each state, at most $|\Sigma' \cup V|$ transitions are added, so this step takes time $O(|V|)$ and space $O(|V|)$. The resulting automaton still has $O(1)$ states. Finally, by inverting the set of final states, we get its complement with the same number of states and transitions, in constant time and $O(|V|)$ space.

This whole procedure therefore constructs in time $O\left(|V|^2\right)$ and space $O(|V|)$ an automaton A'' with $O(1)$ states, which recognizes $\overline{(\Sigma' \cup V)^* . lhs(\rightarrow) . (\Sigma' \cup V)^*}$.

Finally, computing the intersection on alphabet $\Sigma' \cup V$ of two automata with respectively s and s' states takes time $O\left(s^2 \cdot s'^2 \cdot |V|^2\right)$ and space $O\left(s^2 \cdot s'^2 \cdot |V|\right)$ and yields an automaton with $s \cdot s'$ states.

Hence, with $s = |A'| = |A|$ and $s' = |A''| = O(1)$, the intersection of A' and A'' takes time $O\left(|A|^2 \cdot |V|^2\right)$ and space $O\left(|A|^2 \cdot |V|\right)$ and yields an automaton with $O(|A|)$ states.

Adding this all up, since we have $|V| = |R_{\mathcal{B}}|$, we get the following overall time complexity:

$$O\left(|R_{\mathcal{B}}|^2 \cdot |A|^3\right) + O\left(|R_{\mathcal{B}}|^2\right) + O\left(|A|^2 \cdot |R_{\mathcal{B}}|^2\right) .$$

Similarly, the overall space complexity is:

$$O\left(|R_{\mathcal{B}}|^2 \cdot |A|^2\right) + O(|R_{\mathcal{B}}|) + O\left(|A|^2 \cdot |R_{\mathcal{B}}|\right) .$$

Theorem 3. *Let A be a trace automaton and $\mathcal{C} = \{\mathcal{B}_i\}_{1 \leq i \leq n}$ be a set of behavior patterns recognized by regular SRSs $\{R_{\mathcal{B}_i}\}_{1 \leq i \leq n}$. Let $|\mathcal{C}| = \sum_{1 \leq i \leq n} |R_{\mathcal{B}_i}|$.*

Then an automaton of size $O(|A|)$ recognizing $\mathcal{L}(A) \downarrow_{\mathcal{C}} |_{\Gamma}$ can be constructed in time $O\left(|A|^3 \cdot |\mathcal{C}|^2\right)$ and space $O\left(|A|^2 \cdot |\mathcal{C}|^2\right)$.

Proof. Let $R_{\mathcal{B}_i} = (\Sigma, V_i, \rightarrow_i, S_i)$.

Let $V = \bigcup_{1 \leq i \leq n} V_i$ and S a new variable distinct from variables of V.

Let $\rightarrow_{\mathcal{C}} = \bigcup_{1 \leq i \leq n} (\rightarrow_i \cup \{S_i \rightarrow \lambda_i\})$.

Apply the proof of Theorem 5 to the regular SRS:

$$R_{\mathcal{C}} = \left(\Sigma \cup \{\lambda_i\}_{1 \leq i \leq n}, V, \rightarrow_{\mathcal{C}}, S\right) .$$

The number of variables of $R_{\mathcal{C}}$ is precisely $|\mathcal{C}| = \sum_{1 \leq i \leq n} |R_{\mathcal{B}_i}|$ so we construct an automaton A' of size $O(|A|)$ recognizing $\mathcal{L}(A) \downarrow_{\mathcal{C}}$, in time $O\left(|A|^3 \cdot |\mathcal{C}|^2\right)$ and space $O\left(|A|^2 \cdot |\mathcal{C}|^2\right)$.

Finally, projection of A' on Γ is linear in the number of transitions of A', hence the result.

Theorem 4. *Let $\mathcal{D}$ be a set of regular malicious behaviors on Γ, recognized by an automaton $A_{\mathcal{D}}$. Let $\mathcal{M}$ be a machine, with a Γ-abstract trace language recognized by an automaton A. Then deciding whether $\mathcal{M}$ is malicious with respect to $\mathcal{D}$ takes time: $O\left(|A_{\mathcal{D}}|^2 \cdot |A|^2\right)$.*

Proof. Define $A'_{\mathcal{D}}$ to be the automaton built from $A_{\mathcal{D}}$ recognizing the set of words containing a subword in $\mathcal{L}(A_{\mathcal{D}})$ (note that $|A'_{\mathcal{D}}| = O(|A_{\mathcal{D}}|)$).

Then deciding whether t is malicious with respect to $\mathcal{D}$ reduces to testing emptiness of $\mathcal{L}(A'_{\mathcal{D}}) \cap \mathcal{L}(A)$.

Intersecting both automata takes time $O\left(|A'_{D}|^2 \cdot |A|^2\right) = O\left(|A_{\mathcal{D}}|^2 \cdot |A|^2\right)$ and yields an automaton of size $O(|A'_{\mathcal{D}}| \cdot |A|)$. Testing emptiness of the resulting automaton is linear in its size, hence the result.

B Allaple.A example

```
void scan_dir(const char* dir) {
  HANDLE hFind;
  char szFilename[2048];
  WIN32_FIND_DATA findData;

  sprintf(szFilename, "%s\\%s", dir, "*.*");
  hFind = FindFirstFile(szFilename, &findData);
  if (hFind == INVALID_HANDLE_VALUE) return;
  do {
    sprintf(szFilename, "%s\\%s", dir, findData.cFileName);
    if (findData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
      scan_dir(szFilename);
    else { ... }
  } while (FindNextFile(hFind, &findData));
  FindClose(hFind);
}

void main(int argc, char** argv) {
  HANDLE hIcmp;
  const char* icmpData = "Babcdef...";
  char reply[128];

  /* Behavior pattern: ping of a remote host */
  hIcmp = IcmpCreateFile();
  for(int i = 0; i < 2; ++i)
    IcmpSendEcho(hIcmp, ipaddr, icmpData, 10, NULL, reply, 128,
        1000);
  IcmpCloseHandle(hIcmp);

  /* Behavior pattern: Netbios connection */
  SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
  struct sockaddr_in sin = {AF_INET, ipaddr, htons(139)/* Netbios
      */};
  if (connect(s, (SOCKADDR*)&sin, sizeof(sin)) != SOCKET_ERROR) {
    ...
  }

  /* Behavior pattern: scanning of local drives */
  char buffer[1024];
  GetLogicalDriveStrings(sizeof(buffer), buffer);
  const char* szDrive = buffer;
  while (*szDrive) {
    if (GetDriveType(szDrive) == DRIVE_FIXED)
      scan_dir(szDrive);
    szDrive += strlen(szDrive) + 1;
  }
}
```

Fig. 5. Excerpt of the Allaple.A worm

C Experiments

C.1 Behavior patterns with data constraints

In order to yield more relevant data, we define *behavior patterns with data constraints*. These patterns add constraints on function call arguments. For instance, instead of simply detecting a call to `NtOpenFile`, we may want to detect that the opened file is a system file by adding a constraint on the argument representing the file path. Similarly, instead of detecting any access to the registry, we may add a constraint on the key being accessed by requiring that it represents the `Run` key used for malware persistence. Enforcing these constraints is made possible by the capture of call arguments. And more importantly, the extended patterns do not actually alter our formalism since it simply amounts to defining new symbols describing the validation of these constraints: the trace automata are finally modified by adding transitions labelled with these symbols whenever the specified constraints are verified.

C.2 Examples of behavior patterns

We describe some of the behavior patterns used in abstraction:

- writing to system files, which corresponds either to opening a file in a system directory and writing to that file, or to copying a file to a system directory;
- persisting oneself in the Windows registry, which corresponds to adding a subkey in the `Run` registry key or modifying the `AppInit_DLLs` registry subkey;
- browsing drives and files, which corresponds to retrieving the list of drives and scanning their files;
- communicating on IRC, which corresponds to sending IRC messages on a network connection, such as the mandatory "NICK <mynickname>";
- looking for existing antiviruses, by checking registry entries and system services;
- setting up a system hook, using antidebug tricks, enumerating processes, querying access rights, e.g. to check for administrator rights...

C.3 Complete abstraction example of Agent.ah

Figure 6 shows the trace automaton of a thread of the Agent.ah email worm. It is restricted to function calls occurring in the behavior patterns. The trace automaton is then abstracted with respect to our set of 40 behavior patterns. Three of them were matched:

- `TEST_IF_ADMIN`: Tests if the user has administrator rights.
  `TEST_IF_ADMIN = IsNtAdmin + (OpenThreadToken + OpenProcessToken) . AllocateAndInitializeSid [nSubAuthorityCount = 2, dwSubAuthority0 = 0x20, dwSubAuthority1 = 0x220] . AccessCheck`

– `ACQUIRE_DEBUG_PRIVILEGES`: Asks for debugging privileges, in order to access process space of the running processes.
`ACQUIRE_DEBUG_PRIVILEGES = LookupPrivilegeValue [lpName = ''SeDebug Privilege''] . (NtAdjustPrivilegesToken [NewState->Privileges[0]. Attributes = 2] + AdjustTokenPrivileges [NewState->Privileges[0]. Attributes = 2])`
– `ENUM_PROCESSES`: enumerate running processes.
`ENUM_PROCESSES = EnumProcesses + CreateToolhelp32Snapshot [dwFlags & 2 != 0] . Process32First`

The resulting Γ-abstract trace automaton is shown in figure 7.

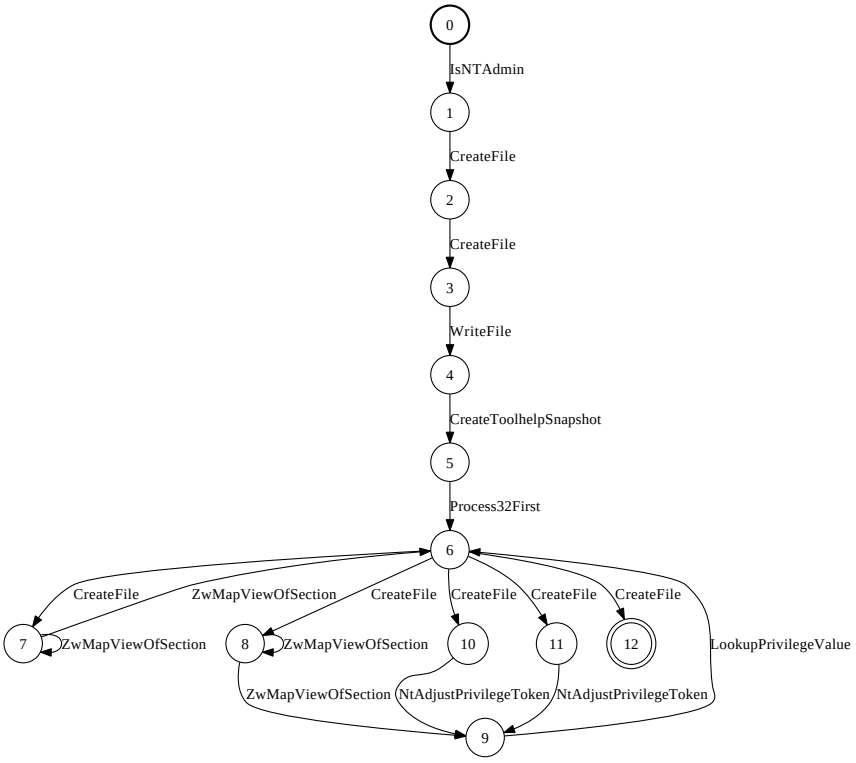

Fig. 6. Agent.ah partial trace automaton

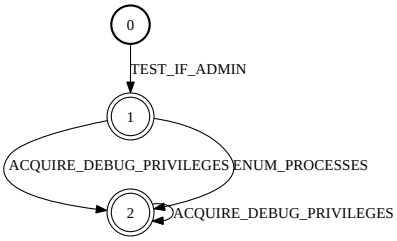

Fig. 7. Agent.ah Γ-abstract trace automaton