On behavioral detection

Philippe Beaucamps and Jean-Yves Marion

{Philippe.Beaucamps,Jean-Yves.Marion}@loria.fr
Nancy-Université - INPL
LORIA
B.P. 239, 54506 Vandœuvre-lès-Nancy Cédex, France

Abstract. This study is about behavioral detection based on automata over infinite words. Malware are considered as concurrent systems, which interact with an environment. So malware traces are now infinite words. We propose a NLOGSPACE behavioral detection method based on Büchi automata. The goal of this paper is to present in a nutshell some theoretical aspects behind behavioral analysis. We don't take up questions related to implementations, which will be studied in forthcoming papers.

1 Introduction

Today, systems are now inter-connected, and programs are mobile. Our point of view is that we can not consider a malware as a sequential program, but rather as a system which interacts with its environment. With ubiquitous computing comes ubiquitous malware ! Since the early days and the seminal work [4] of Cohen in 1986, behavioral detection is a known tool to detect malicious programs. Most of the approaches are function-based. That is, it consists in determining whether or not system calls are legitimate. We refer to the very nice survey [8] for a rather complete overview of all the different aspects and challenges of behavioral detection.

One of the main advantages of behavioral detection compared to traditional syntactic approaches is the fact that it can handle some mutations by obfuscations, which will become soon necessary [6]. However, the study [7] shows that the current implementations are not fully operational. Nevertheless, there are several reasons to push behavioral detection. In particular, it may help new detection methods like the one described in a series of papers [3, 13] and also the one presented in [2], combining syntactic and semantic aware detectors. In this last paper, a new method named *morphological analysis* is proposed, developed and tested. Detection is based on the recognition of an abstract malware control flow graph. We do think that behavioral analysis may help to improve the construction of abstract malware control flow graphs.

In this paper, we propose to deal with the general notion of program traces in order to detect malware. The main novelty is that we consider traces as infinite words over a finite alphabet. Indeed, a worm, which is scanning some channels, has no reason to terminate. To deal with infinite traces, we use the classical theory of automata on infinite words [14]. We show how to use Büchi

automata to detect malicious behaviors. This gives a NLOGSPACE detection method. From this, we try to propose a definition of how to represent a malware behavior based on automata which has the main advantage to open the door to automatic methods coming from model-checking.

1.1 Prerequisite

Suppose that Σ is a finite alphabet. The set Σ^* is the set of finite words over Σ and Σ^ω is the set of infinite words over Σ. An infinite word $u \in \Sigma^\omega$ is written u=u(0)u(1)...u(i)...where each $u(i)$ is in Σ. We set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$.

2 From semantics to finite trace detections

2.1 Syntax and Semantics of a core `WHILE` language

We introduce a simple programming language `WHILE`, which has enough expressive power for our study. The objective is to relate the operational semantics to the notion of traces, which plays a key role, as we shall see, in behavioral detection. The programming language `WHILE` is convenient to illustrate pour point of view. The syntax of `WHILE` is given by the following grammar

$$\text{Expressions: } \mathit{Exp} \rightarrow \mathit{Var} \mid \mathit{Label} \mid \mathit{Exp}\ \texttt{Bop}\ \mathit{Exp} \mid \texttt{Uop}\ \mathit{Exp}$$

$$\begin{aligned}\text{Commands: } \mathit{Stm} \rightarrow\ & \mathit{Label} : \mathit{Var} := \mathit{Exp} \mid \mathit{Stm}_1 ; \mathit{Stm}_2 \mid \mathit{Label} : \texttt{while}(\mathit{Exp})\{\mathit{Stm}\} \mid \\ & \mathit{Label} : \texttt{if}(\mathit{Exp})\{\mathit{Stm}_1\}\texttt{else}\{\mathit{Stm}_2\} \mid \texttt{Call}\mathit{Exp}\end{aligned}$$

Here *Var* denotes a variable, which corresponds to some memory location. *Label* is a program location, which also corresponds to a memory location. `Uop` and `Bop` correspond to respectively unary operators and binary operators. In particular, commands `Call`*Exp* allow to make a system calls. Each program statement is identified by a unique label. A `WHILE` program **p** is just a statement *Stm*. The domain of computation is **Value**. We assume that the evaluation of each expression (and a fortiori of each label) is in **Value**. Both values and locations belong to the same set **Value** and are treated as the same object. Thus, we allow program self-reference, which is an important feature to deal with virus/-worm auto-reproduction and mutation. A modern theoretical foundation of virus mutations may be found in [1].

An operational semantics of the programming language `WHILE` is given by a system environment *Sys* and a store σ. A store $\sigma : \mathbf{Value} \mapsto \mathbf{Value}$ maps a location to a value. A system environment $\mathit{Sys} : \mathbf{Value} \times \mathbf{Store} \mapsto \mathbf{Store}$ returns a store after the execution of a (system) call. Now, suppose one is given a `WHILE` program **p** and a system environment *Sys*. The value of an expression $e \in \mathit{Exp}$ wrt a store σ is given by $[\![e\sigma]\!]$. A *program configuration* is a couple (lb, σ) in which lb is a label and σ a store. A computation is a sequence of configurations, which may be finite if the program **p** halts or infinite.

$$(lb_0, \sigma_0) \rightarrow (lb_1, \sigma_1) \rightarrow \ldots \rightarrow (lb_i, \sigma_i) \rightarrow \ldots$$

This computation depends on the initial configuration. Here the initial configuration is (lb_0, σ_0). It is convenient to define *Configuration* as the set of configurations and $Configuration^{\infty}$ as the set of computations.

In this paper, we focus mostly on infinite computations because a malware is a program, which interacts with its environment and often does not terminate. (In [9], the authors propose a theoretical virus model based on interactions.)

2.2 Traces

A program configuration contains all information about a program state at some time. However, in the context of malware detection, this information is too large. We need therefore to restrict to parts of the informations contained inside a configuration that we call a *trace*. For example, a sequence of system calls may be considered as a trace, or the interactions between processes, or yet some particular sequences of instructions.

Let us focus on system calls. Given a configuration (lb, σ), we define a mapping tr which returns a single trace, as follows:

$$tr(lb, \sigma) = \begin{cases} \texttt{Call}\ e & \text{if } lb : \texttt{Call}\ E \text{ and } e = [\![E\sigma]\!] \\ \epsilon & \text{otherwise} \end{cases}$$

Now, given a computation

$$(lb_0, \sigma_0) \rightarrow (lb_1, \sigma_1) \rightarrow \ldots \rightarrow (lb_i, \sigma_i) \rightarrow \ldots$$

we collect the trace

$$tr(lb_0, \sigma_0) \oplus tr(lb_1, \sigma_1) \oplus \ldots \oplus tr(lb_i, \sigma_i) \oplus \ldots$$

where $\oplus$ is the word concatenation. Let us illustrate this by a concrete example. The worm IIS_Worm was created in 1999 and use a buffer overflow in the Internet Information Service (IIS) of Microsoft. IIS_Worm contains the following code in the main function:

```
while (1) {
 in = accept(s,(sockaddr *)&sout,&soutsize);
 CreateThread(0,0,doweb,&in,0,&threadid);
 }
```

(Here the system call "accept" is a shortcut for `Call`*accept*.) It is an infinite loop which is run as long as the process is alive. This loop listens to requests and creates a new thread for each request by executing the function `doweb`. A full analyse of IIS_Worm can be found in Filiol's book [5]. In our setting, this loop runs two system calls on some arguments. So, we may represent a system call trace by the infinite sequence:

`accept requests CreateThread doweb accept requests CreateThread doweb` ...

where `requests` and `doweb` are the arguments of the system calls and may change in time. Another interesting issue concerns the restriction to a given finite set of system calls and to a given finite set of arguments. In this case, a trace is a word of Σ^∞, finite or infinite, over a finite alphabet Σ. In the IIS_Worm loop example, the argument of the system call `accept` is not fixed unlike the argument of the other system call `CreateThread`. Indeed, the new thread always executes the function `doweb`. So we can consider that some system call arguments are fixed (or partially fixed). We use this knowledge in order to restrict traces to words over a finite alphabet. So, a trace for this example could be then

`accept CreatThread:doweb accept CreateThread:doweb...`

In this example, the trace is an infinite word which is defined over a binary alphabet

$$\Sigma = \{\texttt{accept}, \texttt{CreatThread:doweb}\}$$

We move now towards a formalization of traces. A trace is a finite or infinite word over a finite alphabet Σ which is obtained by the extension of a projection over computations. In other words, we first consider $tr : \mathit{Configuration} \mapsto \Sigma$. A trace is the homomorphic extension tr^∞ of tr from $\mathit{Configuration}^\infty$ to Σ^∞ defined by:

$$tr^\infty(\epsilon) = \epsilon$$
$$tr^\infty((lb_0, \sigma_0) \rightarrow \alpha)) = tr(lb_0, \sigma_0) \oplus tr^\infty(\alpha)$$

2.3 Behavioral detection based on finite traces

Since the seminal work of Cohen [4], behavioral detection is based on finite traces which are words of Σ^* for some alphabet Σ. For example in [11], they loosely use finite automata to detect virus self-replication traces. To illustrate this point, we consider the Xanax virus, which infected mail services, IRC channels and .exe files in 2001. We show below a tiny fragment of the code inside the main function, which infects .exe files (we could also present infections by IRC channels, as they are similar wrt behavioral detection).

```
/* Copy the target to temp file host.tmp */
 strcpy(CopyHost,"host.tmp");
 Copyfile( hostfile ,CopyHost,FALSE);
/* Replace the target by the worm */
    strcpy(Virus,argv [0]);
 Copyfile(FullPath, hostfile ,FALSE);
/* Add target code to code worm */
 AddOrig(CopyHost,hostfile);
 /* Erase temp file */
   _unlink("host.tmp");
```

A trace could be the word

`strcpy:tmp Copyfile strcpy Copyfile AddOrig _unlink`

over a finite alphabet Σ of five letters. If we suppose that this trace is a valid signature, then we can compile into a finite automaton which recognizes finite traces of Σ^*. Next this automaton is used to detect suspect behavior.

In this approach, we actually collect traces with respect to some abstraction (here system calls). These traces play the role of a database of malware behavior signatures. Then we compile them into an automaton that we minimize in order to have a representation of optimal size. Now, given a program **p**, we extract a trace of it and we check whether or not it has a malicious behavior, using to the minimal automaton.

This method is very efficient. However, we may observe that the piece of code above is inside a loop, which searches for files to infect. This loop creates a trace which is potentially unbounded. Moreover, a malware may be seen as an interactive process and so its behavior may be difficult to capture with finite words.

3 Ubiquitous malware

Nowadays, computers are communicating, codes are mobile and this all lives in an ubiquitous world. One of our goals is to scale up malware detection in order to take into account the fact that a malware is not a sequential process anymore, but rather a reactive system. That is why, we try to explore malware detection with infinite traces and we also propose a definition of malware behavior.

3.1 Infinite malware Traces

As we have seen, a trace may be seen as an infinite word. The most simple class of automata, which deals with infinite words, are the Büchi automata. A Büchi automaton $\mathcal{A}$ over the alphabet Σ is defined by a quadruplet $\mathcal{A} = (Q, q_0, \Delta, Q_F)$ where Q is a finite set of states, q_0 is the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $Q_F \subseteq Q$ is the subset of final states.

The IIS_Worm trace that we have discussed in the previous section is recognized by the automaton which is drawn in Figure 1.

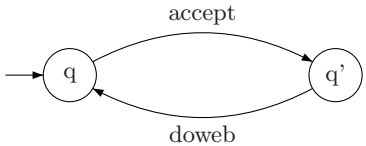

Fig. 1. Trace automaton of IIS_Worm

The behavior of the recent Storm worm can also be modelled by a trace automaton. Figures 2 and 3 give an overview of the trace automaton of Storm, based on the study by Porras, Saïdi and Yegneswaran [12]. Each node in the main automaton, in figure 2, is associated with some sub-automaton like the one in figure 3.

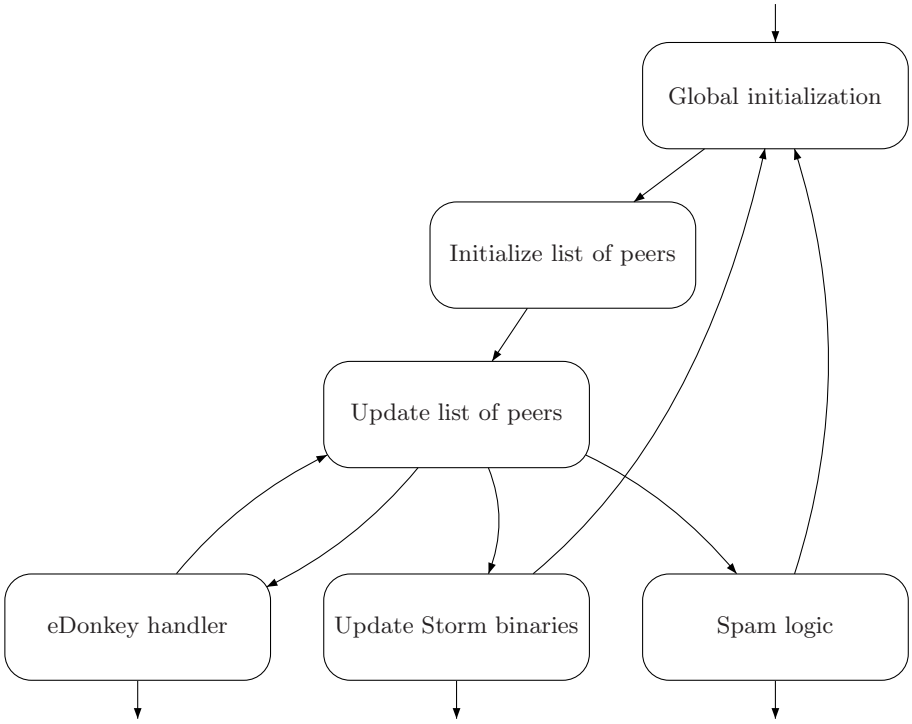

Fig. 2. Overview of the trace automaton of Storm

A run over an infinite word $u = u(1)u(2)\ldots$ is an infinite state sequence $q_0, q_1, \ldots, q_i, q_{i+1}, \ldots$ such that for any i, $(q_i, u(i), q_{i+1}) \in \Delta$. Now, a run is accepting if there is an infinity of final states of Q_F which occur in a run. A Büchi automaton $\mathcal{A}$ accepts (or recognizes) a word u if there is an accepting run on u. Define $\mathcal{L}(\mathcal{A}) = \{u \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } u\}$. A language L is ω-recognizable if there is a Büchi automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

Now, suppose that we have a ω-recognizable language S of traces of malware that we consider as valid signatures. Let $\mathcal{A}_S$ be a Büchi automaton such that $\mathcal{L}(\mathcal{A}_S) = S$. There are several ways to obtain program traces. We can use static analysis. Or we can obtain a trace dynamically by using a virtual environment or by code instrumentation. In the context of this paper, we suppose that we

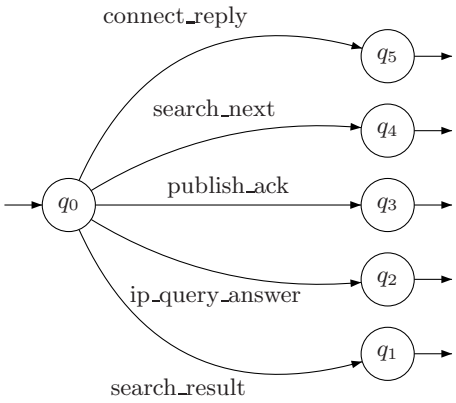

Fig. 3. Detailed automaton of the eDonkey handler of Storm

have access to a trace u of a target program. For example u may be encoded as a stream or we have a symbolic representation of a trace by means of a control flow graph. To check if the trace u contains a signature of S, we may decide if u is accepted by $\mathcal{A}_S$.

3.2 Behavioral detection using infinite traces

Suppose that $\mathbf{p}$ is a target program and we want to analyse it in order to determine whether or not its behavior is similar to the one of a known malware. For this, we have a ω-recognizable language S of infinite malware traces. This set S plays the role of a set of behavioral signatures, which is compiled into a Büchi automaton $\mathcal{A}_S$, as said previously.

Now, a program $\mathbf{p}$ has several traces. Each trace of $\mathbf{p}$ is an execution, which depends on inputs and on its environment. In all cases, here we suppose that we have access to a ω-recognizable language $\mathit{Traces}(\mathbf{p})$ of traces of $\mathbf{p}$. Now, we say that $\mathbf{p}$ behaves as malware wrt S, if there is a trace $u \in \mathit{Traces}(\mathbf{p})$ such that $u \in S$. In other words, $\mathit{Traces}(\mathbf{p}) \cap S \neq \emptyset$.

Theorem 1. *Assume that S is a ω-recognizable language of infinite malware signatures. Assume also that $\mathbf{p}$ is a program and that $\mathit{Traces}(\mathbf{p})$ is a ω-recognizable language of traces of $\mathbf{p}$. There is a NLOGSPACE procedure to decide whether or not $\mathbf{p}$ behaves as a malware.*

The idea of the demonstration is the following. Both, $\mathit{Traces}(\mathbf{p})$ and S are recognized by a Büchi automaton. We have to determine whether $\mathit{Traces}(\mathbf{p}) \cap S$ is empty or not. If it is empty then there is no trace of $\mathbf{p}$ which corresponds to a malware trace. Otherwise, $\mathbf{p}$ has a trace which is similar to a malware trace. Vardi and Wolper [15] show that the non emptiness problem for Büchi automata is logspace-complete for NLOGSPACE. Recall that NLOGSPACE is

the class of problem which is decidable in non-deterministic logarithmic space. NLOGSPACE is included into PTIME.

3.3 Malware behaviors and conclusions

Automata over finite or infinite words can be used to model systems and they are one of the key elements of model checking, see [10] for example. We suggest, and this may be the main contribution of this paper, that the behavior of a malware $\mathcal{M}$ may be specified by a automaton over finite or infinite words $\mathcal{A}_{\mathcal{M}}$. To illustrate this, consider again the IIS_Worm example. The behavior that we have described of IIS_Worm can be represented by an automaton, see Figure 4. So, given a set of malware, we can compile them into a single automaton $\mathcal{A}_S$ since each malware behavior is represented by an automaton and the union of two automata is an automaton. The language $\mathcal{L}(\mathcal{A}_S)$ is the set of malware behaviors. Now, a target program $\mathbf{p}$ can also be represented by an automaton $\mathcal{A}_{\mathbf{p}}$. Next, $\mathbf{p}$ is detected if $\mathcal{L}(\mathcal{A}_{\mathbf{p}}) \cap \mathcal{L}(\mathcal{A}_S)$. This test can be performed efficiently in NLOGSPACE. Lastly, we can also use temporal logic to detect if a program is a malware. In our example, this means that $\mathbf{p}$ behaves as IIS_Worm if $\mathbf{p}$ satisfies the LTL formula $\Box(\texttt{accept} \rightarrow \Diamond\texttt{doweb})$. This formula means that at any time, when a request is sent, then the system will eventually run the thread `doWeb`. It an example of liveness property.

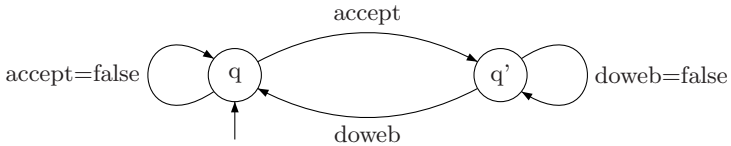

Fig. 4. An automaton speciifcation of IIS_Worm

References

1. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. On abstract computer virology from a recursion-theoretic perspective. *Journal in Computer Virology*, 1(3-4), 2006.
2. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 2008.
3. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, pages 32–46, Oakland, CA, USA, May 2005. ACM Press.
4. F. Cohen. *Computer Viruses*. PhD thesis, University of Southern California, January 1986.
5. E. Filiol. *Computer Viruses: from Theory to Applications*. Springer-Verlag, 2005.
6. E. Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2(1):70–75, 2007.

7. E. Filiol, G. Jacob, and M. L. Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3:23–37, 2007.
8. G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4:251–266, 2008.
9. G. Jacob, E. Filiol, and H. Debar. Malwares as interactive machines: A new framework for behavior modelling. *Journal in Computer Virology*, 4(3):235–250, September 2007.
10. F. Kröger and S. Merz. *Temporal Logic and State Systems*. Springer, 2008.
11. J. Morales, P. Clarke, Y. Deng, and G. Kibria. Characterization of virus replication. *Journal in Computer Virology*, 4(3):221–234, August 2007.
12. P. Porras, H. Saïdi, and V. Yegneswaran. A Multi-perspective Analysis of the Storm (Peacomm) Worm.
13. M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 377–388, New York, NY, USA, Jan. 17–19, 2007. ACM Press.
14. W. Thomas. *Automata on infinite objects*. MIT Press, Cambridge, MA, USA, 1990.
15. M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.