Pushdown Model Checking for Malware Detection

Fu Song and Tayssir Touili

LIAFA, CNRS and Univ. Paris Diderot, France.
E-mail: {song,touili}@liafa.jussieu.fr

Abstract. The number of malware is growing extraordinarily fast. Therefore, it is important to have efficient malware detectors. Malware writers try to obfuscate their code by different techniques. Many of these well-known obfuscation techniques rely on operations on the stack such as inserting dead code by adding useless push and pop instructions, or hiding calls to the operating system, etc. Thus, it is important for malware detectors to be able to deal with the program's stack. In this paper we propose a new model-checking approach for malware detection that takes into account the behavior of the stack. Our approach consists in : (1) Modeling the program using a Pushdown System (PDS). (2) Introducing a new logic, called SCTPL, to represent the malicious behavior. SCTPL can be seen as an extension of the branching-time temporal logic CTL with variables, quantifiers, and predicates over the stack. (3) Reducing the malware detection problem to the model-checking problem of PDSs against SCTPL formulas. We show how our new logic can be used to precisely express malicious behaviors that could not be specified by existing specification formalisms. We then consider the model-checking problem of PDSs against SCTPL specifications. We reduce this problem to emptiness checking in Symbolic Alternating Büchi Pushdown Systems, and we provide an algorithm to solve this problem. We implemented our techniques in a tool, and we applied it to detect several viruses. Our results are encouraging.

1 Introduction

To identify viruses, existing antivirus systems use either code emulation or signature (pattern) detection. These techniques have some limitations. Indeed, emulation based techniques can only check the program's behavior in a limited time interval, whereas signature based systems are easy to get around. To sidestep these limitations, instead of executing the program or making a syntactic check over it, virus detectors need to use analysis techniques that check the *behavior* (not the syntax) of the program in a *static* way, i.e. without executing it. Towards this aim, we propose in this paper to use *model-checking* for virus detection. Model-checking has already been used for virus detection in [7, 23, 12, 14, 19, 18, 20]. However, these works model the program as a finite state graph (automaton). Thus, they are not able to model the stack of the programs, and cannot track the effects of the push, pop and call instructions. However, as decribed in [22], many obfuscation techniques rely on operations over the stack. Indeed, many antivirus systems determine whether a program is malicious by checking the calls it makes to the operating system. Hence, several virus writers try to hide these calls by replacing them by push and return instructions [22]. Therefore, it is important to have analysis techniques that can deal with the program stack.

We propose in this paper a novel model-checking technique for malware detection that takes into account the behavior of the stack. Our approach consists in modeling the program using a *pushdown system* (PDS), and defining a new logic, called *SCTPL*, to express the malicious behavior.

Using pushdown systems as program model allows to consider the program stack. In our modeling, the PDS control locations correspond to the program's control points, and the PDS stack mimics the program's execution stack. This allows the PDS to mimic the behavior of the program. This is different from standard program translations to PDSs where the control points of the program are stored in the stack [16, 6]. These standard translations assume that the program follows a standard compilation model, where the return addresses are never modified. We do not make such assumptions since behaviors where the return addresses are modified can occur in malicious code. We only make the assumption that pushes and pops can be done only using *push*, *pop*, *call*, and *return* operations, not by manipulating the stack pointer.

The logic SCTPL that we introduce is an extension of the CTPL logic that allows to use predicates over the stack. CTPL was introduced in [19, 18, 20]. It can be seen as an extension of CTL with variables and quantifiers. In CTPL, propositions can be predicates of the form $p(x_1, \ldots, x_n)$, where the x_i's are free variables or constants. Free variables can get their values from a finite domain. Variables can be universally or existentially quantified. CTPL is as expressive as CTL, but it allows a more succinct specification of the malicious behavior. For example, consider the statement "The value *data* is assigned to some register, and later, the content of this register is pushed onto the stack." This statement can be expressed in CTL as a large formula enumerating all the possible registers:

$$EF\ (mov(eax, data) \wedge AF\ push(eax)) \vee$$
$$EF\ (mov(ebx, data) \wedge AF\ push(ebx)) \vee$$
$$EF\ (mov(ecx, data) \wedge AF\ push(ecx)) \vee \ldots$$

where every instruction is regarded as a predicate, i.e., $mov(eax, data)$ is a predicate. However, the CTL formula is large for such a simple statement. Using CTPL, this can be expressed by the CTPL formula $\exists r\ EF\ (mov(r, data) \wedge AF\ push(r))$ which expresses in a succinct way that there exists a *register r* such that the above holds. [19, 18, 20] show how this logic is adequate to specify some malicious behaviors. However, CTPL does not allow to specify properties about the stack (which is important for malicious code detection as explained above).

For example, consider Figure 1(a). It corresponds to a critical fragment of the Email-worm Avron [17] that shows the typical behavior of an email worm: it calls an API function *GetModuleHandleA* with 0 as its parameter. This allows to get the entry address of its own executable so that later, it can infect other files by copying this executable into them. (Parameters to a function in assembly are passed by pushing them onto the stack before a call to the function is made. The code in the called function later retrieves these parameters from the stack.) Using CTPL, we can specify this malicious behavior by the following formula:

```
l1: mov eax,0
l2: push eax
l3: call ds:GetModuleHandleA
```

(a)

```
l'1: mov eax,0
l'2: push eax
l'3: push ebx
l'4: pop ebx
l'5: call ds:GetModuleHandleA
```

(b)

Fig. 1. (a) Worm fragment; (b) Obfuscated fragment.

$$\exists\, r_1\ \mathbf{EF}\ \Big(mov(r_1, 0) \wedge \mathbf{EX}\ \mathbf{E}[\neg\exists r_2\ mov(r_1, r_2)\ \mathbf{U}\ (push(r_1) \wedge$$
$$\mathbf{EX}\ \mathbf{E}[\neg\exists\ r_3\ (push(r_3) \vee pop(r_3))\ \mathbf{U}\ call(GetModuleHandleA)]])\Big). \quad (1)$$

This formula states that there exists a register r_1 assigned by 0 such that the value of r_1 is not modified until it is pushed onto the stack. Later the stack is not changed until function *GetModuleHandleA* is called. This specification can detect the fragment in Figure 1(a). However, a worm writer can easily use some obfuscation techniques in order to escape this specification. For example, let us introduce one push followed by one pop after *push eax* at line l_2 as done in Figure 1(b). By doing so, this fragment keeps the same malicious behavior than the fragment in Figure 1(a). However, it cannot be detected by the above CTPL formula. Since the number of pushes and pops that can be added by the worm writer can be arbitrarily large, it is always possible for worm developers to change their code in order to escape a given CTPL formula.

To overcome this problem, we introduce the SCTPL logic which extends CTPL by predicates over the stack. Such predicates are given by regular expressions over the stack alphabet and some free variables (which can also be existantially and universally quantified). Using our new logic SCTPL, the malicious behavior of Figures 1(a) and (b) can be specified as follows:

$$\psi = \exists r_1\ \mathbf{EF}\Big(mov(r_1,0) \wedge \mathbf{EX}\ \mathbf{E}[\neg\exists r_2\ mov(r_1,r_2)\mathbf{U}\ (push(r_1) \wedge \mathbf{EX}\ \mathbf{E}[\neg\Big(push(r_1) \vee(\exists r_3(pop(r_3) \wedge r_1\Gamma^*))\Big)\mathbf{U}\ (call(GetModuleHandleA) \wedge r_1\Gamma^*)])]\Big) \quad (2)$$

where $r_1\Gamma^*$ is a regular predicate expressing that the topmost symbol of the stack is r_1. The SCTPL formula ψ states that there exists a register r_1 assigned by 0 such that the value of r_1 is not changed until it is pushed onto the stack. Then, r_1 is never pushed onto the stack again nor popped from it until the function *GetModuleHandleA* is called. When this call is made, the topmost symbol of the stack has to be r_1. This ensures that *GetModuleHandleA* is called with 0 as parameter. This specification can detect both fragments in Figure 1, because it allows to specify the content of the stack when *GetModuleHandleA* is called. Note that it is important to use pushdown systems as model in order to have specifications with predicates over the stack.

The main contributions of this paper are:

1. We present a new technique to translate a binary program into a pushdown system that mimics the program's behavior (a malicious program is usually an executable, i.e., a binary program). Our translation is different from standard program translations to PDSs that need to assume that the program follows a standard compilation model, where the return addresses are never modified. Our translation does not need to make this assumption since malicious code may have a non standard form.
2. We introduce the SCTPL logic and show how it can be used to efficiently and precisely characterize different malicious behaviors.
3. We propose an algorithm for model checking pushdown systems against SCTPL specifications. We reduce this problem to checking emptiness in Symbolic Alternating Büchi Pushdown Systems (SABPDS), and we propose an algorithm to solve this emptiness problem.
4. We implemented our techniques in a tool that we successfully applied to detect several viruses.

Related work. Model-checking and static analysis techniques have been applied to detect malicious behaviors e.g. in [7, 23, 12, 14, 19, 18, 20]. However, all these works are based on modeling the program as a finite-state system, and thus, they miss the behavior of the stack. As we have seen, being able to track the stack is important for many malicious behaviors. [8] use tree automata to represent a set of malicious behaviors. However, [8] cannot specify predicates over the stack content.

[22] keeps track of the stack by computing an abstract stack graph which finitely represents the infinite set of all the possible stacks for every control point of the program. Their technique can detect only obfuscated calls and obfuscated returns. Using SCTPL, we are able to detect more malicious behaviors.

[21] performs context-sensitive analysis of *call* and *ret* obfuscated binaries. They use abstract interpretation to compute an abstraction of the stack. We believe that our techniques are more precise since we do not abstract the stack. Moreover, the techniques of [21] were only tried on toy examples, they have not been applied for malware detection.

[6] uses pushdown systems for binary code analysis. However, [6] has not been applied for malware detection. Moreover, the translation from programs to PDSs in [6] assumes that the program follows a standard compilation model where calls and returns match. Several malicious behaviors do not follow this model. Our translation from a control flow graph to a PDS does not make this assumption.

[13] defines a language for specifying malicious behavior in terms of dependencies between system calls. Compared to SCTPL, the specification language of [13] does not take the stack into account and is only able to express safety properties (no CTL like properties), whereas SCTPL does. On the other hand, [13] is able to automatically derive the malicious specifications by comparing the execution behavior of a known malware against the execution behaviors of a set of benign programs. It would be interesting to see if their techniques can be extended to automatically derive SCTPL specifications of malicious behaviors.

LTL or CTL model-checking with regular predicates over the stack was considered in [15, 24]. These works do not consider variables and quantifiers.

Outline. We give our formal model in Section 2. In Section 3, we introduce our SCTPL logic. Our SCTPL model checking algorithm for pushdown systems is given in Section 4. The experiments we made for malware detection are reported in Section 5. Due to lack of space, some illustrating examples and proofs are omitted and given in the appendix.

2 Formal model: Pushdown Systems

We model a binary code by a pushdown system (PDS). In our modeling, the PDS control locations correspond to the program's control points, and the PDS stack mimics the program's execution stack. This is different from standard program translations to PDSs where the control points of the program are stored in the stack [16, 6]. These standard translations assume that the program follows a standard compilation model, where the return addresses are never modified. We do not make such assumptions since behaviors where the return addresses are modified can occur in malicious code. We only make the assumption that pushes and pops can be done only using *push*, *pop*, *call*, and *return* operations, not by manipulating the stack pointer. Due to lack of space, we give the details of our translation in Appendix B.

Formally, a *Pushdown System* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, where P is a finite set of control locations, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules, and $\sharp \in \Gamma$ is the bottom stack symbol. A configuration of $\mathcal{P}$ is $\langle p, \omega \rangle$, where $p \in P$ and $\omega \in \Gamma^*$. If $((p, \gamma), (q, \omega)) \in \Delta$, we write $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$. For technical reasons, we assume that the bottom stack symbol $\sharp$ is never popped from the stack, i.e., there is no transition rule of the form $\langle p, \sharp \rangle \hookrightarrow \langle q, \omega \rangle \in \Delta$.

The successor relation $\leadsto_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \leadsto_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$. For every configuration $c, c' \in P \times \Gamma^*$, c is a successor of c' iff $c \leadsto_{\mathcal{P}} c'$. A path is a sequence of configurations $c_0, c_1, ...$ s.t. $c_i \leadsto_{\mathcal{P}} c_{i+1}$ for every $i \geq 0$.

3 Malicious behavior Specification

In this section, we introduce the Stack Computation Tree Predicate Logic (SCTPL), the formalism we use to specify malicious behavior.

3.1 Environments, predicates and regular expressions

From now on, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, ...\}$ be a finite set of variables ranging over a finite domain $\mathcal{D}$. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment function that assigns a value $c \in \mathcal{D}$ to each variable $x \in \mathcal{X}$, and such that $B(c) = c$ for every $c \in \mathcal{D}$. $B[x \leftarrow c]$ denotes the environment function such that $B[x \leftarrow c](x) = c$ and $B[x \leftarrow c](y) = B(y)$ for every $y \neq x$. $Abs_x(B)$ is the set of all the environments B' s.t. for every $y \neq x$, $B'(y) = B(y)$. Let $\mathcal{B}$ be the set of all the environment functions.

Let $AP = \{a, b, c, ...\}$ be a finite set of atomic propositions, $AP_{\mathcal{X}}$ be a finite set of atomic predicates of the form $b(\alpha_1, ..., \alpha_m)$ such that $b \in AP$, $\alpha_i \in \mathcal{X} \cup \mathcal{D}$ for every i, $1 \leq i \leq m$, and $AP_{\mathcal{D}}$ be a finite set of atomic predicates of the form $b(\alpha_1, ..., \alpha_m)$ such that $b \in AP$ and $\alpha_i \in \mathcal{D}$ for every i, $1 \leq i \leq m$.

Let $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ be a PDS s.t. $\Gamma \subseteq \mathcal{D}$. Let $\mathcal{R}$ be a finite set of regular variable expressions e over $\mathcal{X} \cup \Gamma$ defined by:

$$e ::= \emptyset \mid \epsilon \mid a \in \mathcal{X} \cup \Gamma \mid e + e \mid e \cdot e \mid e^*$$

The language $L(e)$ of a regular variable expression e is a subset of $P \times \Gamma^* \times \mathcal{B}$ defined inductively as follows: $L(\emptyset) = \emptyset$; $L(\epsilon) = \{(\langle p, \epsilon\rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(x)$, where $x \in \mathcal{X}$ is the set $\{(\langle p, \gamma\rangle, B) \mid p \in P, \gamma \in \Gamma, B \in \mathcal{B} : B(x) = \gamma\}$; $L(\gamma)$, where $\gamma \in \Gamma$ is the set $\{(\langle p, \gamma\rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(e_1 + e_2) = L(e_1) \cup L(e_2)$; $L(e_1 \cdot e_2) = \{(\langle p, \omega_1\omega_2\rangle, B) \mid (\langle p, \omega_1\rangle, B) \in L(e_1);\ (\langle p, \omega_2\rangle, B) \in L(e_2)\}$; and $L(e^*) = \{(\langle p, \omega^*\rangle, B) \mid (\langle p, \omega\rangle, B) \in L(e)\}$. E.g., $(\langle p, \gamma_1\gamma_1\gamma_2\rangle, B)$ is an element of $L(x^*\gamma_2)$ when $B(x) = \gamma_1$.

3.2 Stack Computation Tree Predicate Logic

We are now ready to define our new logic SCTPL. Intuitively, a SCTPL formula is a CTL formula where predicates and regular variable expressions are used as atomic propositions. Using regular variable expressions allows to express predicates on the stack content of the PDS. Moreover, since predicates and regular variable expressions contain variables, we allow quantifiers over variables. For technical reasons, we suppose w.l.o.g. that formulas are given in positive normal form, i.e., negations are applied only to atomic propositions. Indeed, each CTL formula can be written in positive normal form by pushing the negations inside. Moreover, we use the operator $\tilde{U}$ as a dual of the until operator for which the stop condition is not required to occur. Then, standard CTL operators can be expressed as follows: $EF\psi = E[trueU\psi]$, $AF\psi = A[trueU\psi]$, $EG\psi = E[false\tilde{U}\psi]$ and $AG\psi = A[false\tilde{U}\psi]$.

More precisely, the set of *SCTPL formulas* is given by (where $x \in \mathcal{X}$, $a(x_1, ..., x_n) \in AP_{\mathcal{X}}$ and $e \in \mathcal{R}$):

$$\begin{aligned}\varphi ::= {} & a(x_1, ..., x_n) \mid \neg a(x_1, ..., x_n) \mid e \mid \neg e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x\, \varphi \\ & \mid \exists x\, \varphi \mid AX\varphi \mid EX\varphi \mid A[\varphi U \varphi] \mid E[\varphi U \varphi] \mid A[\varphi \tilde{U} \varphi] \mid E[\varphi \tilde{U} \varphi]\end{aligned}$$

Let φ be a SCTPL formula. The closure $cl(\varphi)$ denotes the set of all the subformulas of φ including φ. The size $|\varphi|$ of φ is the number of elements of $cl(\varphi)$. Let $AP^+(\varphi) =$

$\{a(x_1,...,x_n) \in AP_{\mathcal{X}} \mid a(x_1,...,x_n) \in cl(\varphi)\}$, $AP^-(\varphi) = \{a(x_1,...,x_n) \in AP_{\mathcal{X}} \mid \neg a(x_1,...,x_n) \in cl(\varphi)\}$, $Reg^+(\varphi) = \{e \in \mathcal{R} \mid e \in cl(\varphi)\}$, $Reg^-(\varphi) = \{e \in \mathcal{R} \mid \neg e \in cl(\varphi)\}$, and $cl_{\tilde{U}}(\varphi)$ be the set of formulas of $cl(\varphi)$ in the form of $E[\varphi_1 \tilde{U} \varphi_2]$ or $A[\varphi_1 \tilde{U} \varphi_2]$.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ s.t. $\Gamma \subseteq \mathcal{D}$, let $\lambda : AP_{\mathcal{D}} \to 2^P$ be a labeling function that assigns a set of control locations to a predicate. Let $c = \langle p, w \rangle$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SCTPL formula ψ in c, denoted by $c \models_\lambda \psi$, iff there exists an environment $B \in \mathcal{B}$ s.t. $c \models_\lambda^B \psi$, where $c \models_\lambda^B \psi$ is defined by induction as follows:

- $c \models_\lambda^B a(x_1,...,x_n)$ iff $p \in \lambda\Big(a(B(x_1),...,B(x_n))\Big)$.
- $c \models_\lambda^B \neg a(x_1,...,x_n)$ iff $p \notin \lambda\Big(a(B(x_1),...,B(x_n))\Big)$.
- $c \models_\lambda^B e$ iff $(c, B) \in L(e)$.
- $c \models_\lambda^B \neg e$ iff $(c, B) \notin L(e)$.
- $c \models_\lambda^B \psi_1 \wedge \psi_2$ iff $c \models_\lambda^B \psi_1$ and $c \models_\lambda^B \psi_2$.
- $c \models_\lambda^B \psi_1 \vee \psi_2$ iff $c \models_\lambda^B \psi_1$ or $c \models_\lambda^B \psi_2$.
- $c \models_\lambda^B \forall x\, \psi$ iff $\forall v \in \mathcal{D},\ c \models_\lambda^{B[x \leftarrow v]} \psi$.
- $c \models_\lambda^B \exists x\, \psi$ iff $\exists v \in \mathcal{D}\ s.t.\ c \models_\lambda^{B[x \leftarrow v]} \psi$.
- $c \models_\lambda^B AX\, \psi$ iff $c' \models_\lambda^B \psi$ for every successor c' of c.
- $c \models_\lambda^B EX\, \psi$ iff there exists a successor c' of c s.t. $c' \models_\lambda^B \psi$.
- $c \models_\lambda^B A[\psi_1 U \psi_2]$ iff for every path $\pi = c_0, c_1, ...,$ of $\mathcal{P}$ with $c_0 = c$, $\exists i \geq 0$ s.t. $c_i \models_\lambda^B \psi_2$ and $\forall 0 \leq j < i : c_j \models_\lambda^B \psi_1$.
- $c \models_\lambda^B E[\psi_1 U \psi_2]$ iff there exists a path $\pi = c_0, c_1, ...,$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\exists i \geq 0,\ c_i \models_\lambda^B \psi_2$ and $\forall 0 \leq j < i, c_j \models_\lambda^B \psi_1$.
- $c \models_\lambda^B A[\psi_1 \tilde{U} \psi_2]$ iff for every path $\pi = c_0, c_1, ...,$ of $\mathcal{P}$ with $c_0 = c$, $\forall i \geq 0$ s.t. $c_i \not\models_\lambda^B \psi_2$, $\exists 0 \leq j < i$ s.t. $c_j \models_\lambda^B \psi_1$.
- $c \models_\lambda^B E[\psi_1 \tilde{U} \psi_2]$ iff there exists a path $\pi = c_0, c_1, ...,$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\forall i \geq 0$ s.t. $c_i \not\models_\lambda^B \psi_2$, $\exists 0 \leq j < i$ s.t. $c_j \models_\lambda^B \psi_1$.

Intuitively, $c \models_\lambda^B \psi$ holds iff the configuration c satisfies the formula ψ under the environment B. Note that a path π satisfies $\psi_1 \tilde{U} \psi_2$ iff either ψ_2 holds everywhere in π, or the first occurrence in the path where ψ_2 does not hold must be preceeded by a position where ψ_1 holds.
Example: In Appendix C, we give an example that illustrates the above definitions.

Remark 1. CTPL [20] is a subclass of SCTPL where predicates over the stack are not allowed (i.e., SCTPL formulas that do not use regular variable expressions). SCTPL is more expressive than CTPL since it allows to express predicates over the content of the stack using regular languages.

Remark 2. CTL with regular valuations is an extended version of CTL where the atomic propositions can be regular sets of configurations over the stack alphabet. Since the domain $\mathcal{D}$ is finite, every SCTPL formula ψ can be transformed to an equivalent CTL formula with regular valuations ψ'. This transformation can be done by enumerating all the possible valuations of the variables $\mathcal{X}$. Intuitively, a SCTPL formula $\exists x a(x)$ is equivalent to $\bigvee_{c \in \mathcal{D}} a(c)$, and $\forall x a(x)$ is equivalent to $\bigwedge_{c \in \mathcal{D}} a(c)$. The obtained formula has size $|\psi'| = O(|\psi||\mathcal{D}|^g)$ where g is the number of subformulas of ψ in the form of $\forall x \psi$ or $\exists x \psi$. Thus, SCTPL allows to be more succinct than CTL with regular valuations.

Modeling malicious behaviors using SCTPL: In Appendix D, we show some examples that illustrate how SCTPL can be used to precisely specify malicious behaviors. We needed stack predicates to express most of the specifications. Thus, SCTPL is necessary to specify these behaviors, CTPL is not sufficient.

4 SCTPL Model-Checking for Pushdown Systems

In this section, we give an efficient SCTPL model checking algorithm for Pushdown systems. Our procedure works as follows: we reduce this model checking problem to the emptiness problem in Symbolic Alternating Büchi Pushdown Systems (SABPDS), and we give an algorithm to solve this emptiness problem. To achieve this reduction, we use *variable automata* to represent regular variable expressions. This section is structured as follows. First, we introduce variable automata. Then, we define Symbolic Alternating Büchi Pushdown Systems. Next, we show how SCTPL model checking for PDSs can be reduced to emptiness checking of SABPDSs. Finally, we give an algorithm that solves this problem.

In the remainder of this section, we let $\mathcal{X}$ be a finite set of variables ranging over a finite domain $\mathcal{D}$, and $\mathcal{B}$ be the set of all the environment functions $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$.

4.1 Variable Automata

Given a PDS $\mathcal{P} = (P, \varGamma, \varDelta, \sharp)$ s.t. $\varGamma \subseteq \mathcal{D}$, a *Variable Automaton* (VA) is a tuple $M = (Q, \varGamma, \delta, q_0, A)$, where Q is a finite set of states; $\varGamma$ is the input alphabet; $q_0 \subseteq Q$ is an initial state; $A \subseteq Q$ is a finite set of accepting states; and δ is a finite set of transition rules of the form: $p \xrightarrow{\alpha} \{q_1, ..., q_n\}$ where α can be x, $\neg x$, or γ, for any $x \in \mathcal{X}$ and $\gamma \in \varGamma$.

Let $B \in \mathcal{B}$. A run of VA on a word $\gamma_1, ..., \gamma_m$ under B is a tree of height m whose root is labelled by the initial state q_0, and each node at depth k labelled by a state q has h children labelled by $p_1, ..., p_h$, respectively, such that: either $q \xrightarrow{\gamma_k} \{p_1, ..., p_h\} \in \delta$ and $\gamma_k \in \varGamma$; or $q \xrightarrow{x} \{p_1, ..., p_h\} \in \delta$, $x \in \mathcal{X}$ and $B(x) = \gamma_k$; or $q \xrightarrow{\neg x} \{p_1, ..., p_h\} \in \delta$, $x \in \mathcal{X}$ and $B(x) \neq \gamma_k$. A branch of the tree is accepting iff the leaf of the branch is an accepting state. A run is accepting iff all its branches are accepting. A word $\omega \in \varGamma^*$ is accepted by a VA under an environment $B \in \mathcal{B}$ iff the VA has an accepting run on the word ω under the environment B. The language of a VA M, denoted by $L(M)$, is a subset of $(P \times \varGamma^*) \times \mathcal{B}$. $(\langle p, \omega \rangle, B) \in L(M)$ iff M accepts the word ω under the environment B. We can show that:

Theorem 1. *VAs are effectively closed under boolean operations.*

Theorem 2. *For every regular expression $e \in \mathcal{R}$, one can effectively compute in polynomial time a VA M such that $L(M) = L(e)$.*

4.2 Symbolic Alternating Büchi Pushdown Systems

Definition 1. *A* Symbolic Alternating Büchi Pushdown System *(SABPDS) is a tuple $\mathcal{BP} = (P, \varGamma, \varDelta, F)$, where P is a finite set of control locations; $\varGamma \subseteq \mathcal{D}$ is the stack alphabet; $F \subseteq P \times 2^{\mathcal{B}}$ is a set of accepting states; $\varDelta$ is a finite set of transitions of the form $\langle p, \gamma \rangle \overset{\mathfrak{R}}{\hookrightarrow} [\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle]$ where $p \in P$, $\gamma \in \varGamma$, for every i, $1 \leq i \leq n$: $p_i \in P$, $\omega_i \in \varGamma^*$, and $\mathfrak{R} : (\mathcal{B})^n \longrightarrow 2^{\mathcal{B}}$ is a function that maps a tuple of environments to a set of environments.*

A configuration of a SABPDS is a tuple $\langle [p, B], \omega \rangle$, where $p \in P$ is a control location, $B \in \mathcal{B}$ is an environment and $\omega \in \varGamma^*$ is the stack content. $[p, B] \in P \times \mathcal{B}$ is an accepting state iff $\exists [p, \beta] \in F$ s.t. $B \in \beta$. Let $t = \langle p, \gamma \rangle \overset{\mathfrak{R}}{\hookrightarrow} [\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle] \in \varDelta$ be a transition, n is the width of the transition t. For every $\omega \in \varGamma^*$, $B, B_1, ..., B_n \in \mathcal{B}$, if $B \in \mathfrak{R}(B_1, ..., B_n)$, then the configuration $\langle [p, B], \gamma\omega \rangle$ (resp. $\{\langle [p_1, B_1], \omega_1\omega \rangle, ..., \langle [p_n, B_n], \omega_n\omega \rangle\}$) is an immediate predecessor (resp. immediate successor) of $\{\langle [p_1, B_1], \omega_1\omega \rangle, ..., \langle [p_n, B_n], \omega_n\omega \rangle\}$ (resp. $\langle [p, B], \gamma\omega \rangle$). A run ρ of $\mathcal{BP}$ from an initial configuration $\langle [p_0, B_0], \omega_0 \rangle$ is a tree in which the root is labeled by $\langle [p_0, B_0], \omega_0\rangle$, and the other nodes are labeled by elements of $(P \times \mathcal{B}) \times \Gamma^*$. If a node of ρ labeled by $\langle [p, B], \omega\rangle$ has n children labeled by $\langle [p_1 B_1], \omega_1\rangle, ..., \langle [p_n, B_n], \omega_n\rangle$, respectively, then, necessarily, $\langle [p, B], \omega\rangle$ is an immediate predecessor of $\{\langle [p_1, B_1], \omega_1\rangle, ..., \langle [p_n, B_n], \omega_n\rangle\}$ in $\mathcal{BP}$.

A path $c_0 c_1 ...$ of a run ρ is an *infinite* sequence of configurations where c_0 is the root of ρ and for every $i \geq 0$, c_{i+1} is one of the children of the node c_i in ρ. The path is accepting iff it visits infinitely often configurations with accepting states. A run ρ is accepting iff all its paths are accepting. Note that an accepting run has only *infinite* paths. A configuration c is accepted (or recognized) by $\mathcal{BP}$ iff $\mathcal{BP}$ has an accepting run starting from c. The language of $\mathcal{BP}$, denoted by $\mathcal{L}(\mathcal{BP})$, is the set of configurations accepted by $\mathcal{BP}$.

The predecessor functions $Pre_{\mathcal{BP}}$, $Pre^*_{\mathcal{BP}}$ and $Pre^+_{\mathcal{BP}}$: $2^{(P\times\mathcal{B})\times\Gamma^*} \longrightarrow 2^{(P\times\mathcal{B})\times\Gamma^*}$ are defined as follows: $Pre_{\mathcal{BP}}(C) = \{c \in (P \times \mathcal{B}) \times \Gamma^* \mid$ some immediate successor of c is a subset of C$\}$, $Pre^*_{\mathcal{BP}}$ is the reflexive and transitive closure of $Pre_{\mathcal{BP}}$, $Pre_{\mathcal{BP}} \circ Pre^*_{\mathcal{BP}}$ is denoted by $Pre^+_{\mathcal{BP}}$.

SABPDS vs. ABPDS. An *Alternating Büchi Pushdown System* (ABPDS for short) [24] can be seen as a SABPDS such that $\mathcal{X} = \emptyset$, $\mathcal{D} = \{\perp\}$, and every function $\mathfrak{R} : (\mathcal{B})^n \longrightarrow 2^{\mathcal{B}}$ is of the form $\mathfrak{R}(B_1, ..., B_n) = B_\perp$, where $B_\perp(\perp) = \perp$. Such a function will be denoted by $\mathfrak{R}_\perp$. SABPDSs can be simulated by ABPDSs. Indeed, each SABPDS rule of the form $\langle p, \gamma\rangle \overset{\mathfrak{R}}{\hookrightarrow} [\langle p_1, \omega_1\rangle, ..., \langle p_n, \omega_n\rangle] \in \Delta$ can be translated into a set of ABPDS rules of the form $\langle [\![p, B]\!], \gamma\rangle \overset{\mathfrak{R}_\perp}{\hookrightarrow} [\langle [\![p_1, B_1]\!], \omega_1\rangle, ..., \langle [\![p_n, B_n]\!], \omega_n\rangle]$ where $B, B_1, ..., B_n$ can be any elements in $\mathcal{B}$ s.t. $B \in \mathfrak{R}(B_1, ..., B_n)$. However, this translation is expensive since the number of environments in $\mathcal{B}$ is large:

Lemma 1. *Given a SABPDS $\mathcal{BP} = (P, \Gamma, \Delta, F)$, one can compute an equivalent ABPDS $\mathcal{BP}'$ that simulates $\mathcal{BP}$ in $O(|\Delta| \cdot |\mathcal{B}|^{k+1})$ time, where k is the maximum of the widths of the transition rules in Δ and $|\mathcal{B}| = |D|^{|\mathcal{X}|}$.*

Symbolic Alternating Multi-Automata. To finitely represent infinite sets of configurations of SABPDSs, we use Symbolic Alternating Multi-Automata.

Let $\mathcal{BP} = (P, \Gamma, \Delta, F)$ be a SABPDS, a *Symbolic Alternating Multi-Automaton* (SAMA) is a tuple $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$, where Q is a finite set of states, Γ is the input alphabet, $\delta \subseteq (Q \times \Gamma) \times 2^Q$ is a finite set of transition rules, $I \subseteq P \times 2^{\mathcal{B}}$ is a finite set of initial states, $Q_f \subseteq Q$ is a finite set of final states. An *Alternating Multi-Automaton* (AMA) is a SAMA such that $I \subseteq P \times \{\emptyset\}$.

We define the reflexive and transitive transition relation $\longrightarrow_\delta \subseteq (Q \times \Gamma^*) \times 2^Q$ as follows: (1) $q \xrightarrow{\epsilon}_\delta \{q\}$ for every $q \in Q$, where ϵ is the empty word, (2) if $q \xrightarrow{\gamma} \{q_1, ..., q_n\} \in \delta$ and $q_i \xrightarrow{\omega}_\delta Q_i$ for every $1 \leq i \leq n$, then $q \xrightarrow{\gamma\omega}_\delta \bigcup_{i=1}^n Q_i$. The automaton $\mathcal{A}$ recognizes a configuration $\langle [p, B], \omega\rangle$ iff there exist $Q' \subseteq Q_f$ and $\beta \subseteq \mathcal{B}$ s.t. $B \in \beta$, $[p, \beta] \in I$ and $[p, \beta] \xrightarrow{\omega}_\delta Q'$. The language of $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the set of configurations recognized by $\mathcal{A}$. A set of configurations is regular if it can be recognized by a SAMA. Similarly, AMAs can also be used to recognize (infinite) regular sets of configurations for ABPDSs.

Proposition 1. *Let $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$ be a SAMA. Then, deciding whether a configuration $\langle [p, B], \omega\rangle$ is accepted by $\mathcal{A}$ can be done in $O(|Q| \cdot |\delta| \cdot |\omega| + \tau)$ time, where τ denotes the time used to check whether $B \in \beta$ for some $B \in \mathcal{B}, \beta \subseteq \mathcal{B}$.*

Remark 3. The time τ used to check whether $B \in \beta$ depends on the representation of B and β. In particular, if we use BDDs to represent sets of environment functions, checking whether $B \in \beta$ can be done in $\tau = O(\lceil log|\mathcal{D}|\rceil \cdot |\mathcal{X}|)$ [10].

Examples of functions $\mathfrak{R}$. We give some examples of functions $\mathfrak{R}$ that will be used later.

- $equal(B_1, ..., B_n) = \begin{cases} \{B_1\} \text{ if } B_i = B_j \text{ for every } 1 \leq i, j \leq n, \text{ or } n = 1 \\ \emptyset \quad \text{otherwise.} \end{cases}$
  This function checks that all the B_i's are equal and returns $\{B_1\}$ (which is equal to $\{B_i\}$ for any i) if this is the case and the emptyset otherwise.
- $meet^x_{\{c_1,...,c_n\}}(B_1, ..., B_n) = \begin{cases} Abs_x(B_1) \text{ if } B_i(x) = c_i \text{ and } B_i(y) = B_j(y) \text{ for } y \neq x, \\ \qquad\qquad \text{for every } 1 \leq i, j \leq n, \\ \emptyset \qquad\qquad \text{otherwise.} \end{cases}$
  This function checks whether $B_i(x) = c_i$ for every i, $1 \leq i \leq n$, and for every $y \neq x$ and every i, j, $1 \leq i, j \leq n$ $B_i(y) = B_j(y)$. It returns $Abs_x(B_1)$ (which is equal to $Abs_x(B_i)$ for any i) if this is the case and the emptyset otherwise.
- $join^x_c(B_1, ..., B_n) = \begin{cases} \{B_1\} \text{ if } B_i = B_j \text{ and } B_i(x) = c, \text{ for every } 1 \leq i, j \leq n, \\ \emptyset \quad \text{otherwise.} \end{cases}$
  This function checks whether $B_i(x) = c$ for every i. If this is the case, it returns $equal(B_1, ..., B_n)$, otherwise, it returns the emptyset.
- $join^{\neg x}_c(B_1, ..., B_n) = \begin{cases} \{B_1\} \text{ if } B_i = B_j \text{ and } B_i(x) \neq c, \text{ for every } 1 \leq i, j \leq n, \\ \emptyset \quad \text{otherwise.} \end{cases}$
  This function checks whether $B_i(x) \neq c$ for every i. If this is the case, it returns $equal(B_1, ..., B_n)$, otherwise, it returns the emptyset.

4.3 From SCTPL Model Checking for PDSs to Emptiness of SABPDS

Let $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, $\lambda : AP_{\mathcal{D}} \to 2^P$ be a labeling function, and φ be a SCTPL formula. For every configuration $\langle p, \omega\rangle$, our goal is to determine whether $\langle p, \omega\rangle \models_\lambda \varphi$, i.e., whether there exists an environment $B \in \mathcal{B}$ s.t. $\langle p, \omega\rangle \models^B_\lambda \varphi$. We proceed as follows: we compute a symbolic alternating Büchi pushdown system $\mathcal{BP}$ s.t. $\langle p, \omega\rangle \models^B_\lambda \varphi$ iff $\langle [(\!|p, \varphi|\!), B], \omega\rangle \in \mathcal{L}(\mathcal{BP})$. Then, $\langle p, \omega\rangle \models_\lambda \varphi$ iff there exists $B \in \mathcal{B}$ such that $\langle p, \omega\rangle \models^B_\lambda \varphi$.

Let $Reg^+(\varphi) = \{e_1, ..., e_k\}$ and $Reg^-(\varphi) = \{e_{k+1}, ..., e_m\}$ be the two sets of regular variable expressions[1] that occur in φ. As shown in Theorems 2 and 1, for every i, $1 \leq i \leq k$ we can construct VAs $M_{e_i} = (Q_{e_i}, \Gamma, \delta_{e_i}, s_{e_i}, A_{e_i})$ such that $L(M_{e_i}) = L(e_i)$; and for every j, $k < j \leq m$ we can construct VAs $M_{\neg e_j} = (Q_{\neg e_j}, \Gamma, \delta_{\neg e_j}, s_{\neg e_j}, A_{\neg e_j})$ such that $L(M_{\neg e_j}) = (P \times \Gamma^*) \times \mathcal{B} \setminus L(e_j)$. We suppose w.l.o.g. that the states of these automata are distinct. Let $\mathcal{M}$ be the union of all these automata, $\mathcal{F}$ be the union of all the final states of these automata A_{e_i}'s and $A_{\neg e_j}$'s and $\mathcal{S}$ be the union of all the states of these automata Q_{e_i}'s and $Q_{\neg e_j}$'s.

Let $\mathcal{BP}_\varphi = (P', \Gamma, \Delta', F)$ be the SABPDS defined as follows: $P' = P \times cl(\varphi) \cup \mathcal{S}$; $F = F_1 \cup F_2 \cup F_3 \cup F_4$, where $F_1 = \{[(\!|p, a(x_1, ..., x_n)|\!), \beta] \mid a(x_1, ..., x_n) \in AP^+(\varphi) \text{ and } \beta = \{B \in \mathcal{B} \mid p \in \lambda\big(a(B(x_1), ..., B(x_n))\big)\}\}$; $F_2 = \{[(\!|p, \neg a(x_1, ..., x_n)|\!), \beta] \mid \neg a(x_1, ..., x_n) \in AP^-(\varphi) \text{ and } \beta = \{B \in \mathcal{B} \mid p \notin \lambda\big(a(B(x_1), ..., B(x_n))\big)\}\}$; $F_3 = P \times cl_{\tilde{U}}(\varphi) \times \{\mathcal{B}\}$; and $F_4 = \mathcal{F} \times \{\mathcal{B}\}$.
Δ' is the smallest set of transition rules that satisfy the following. For every control location $p \in P$, every subformula $\psi \in cl(\varphi)$, and every $\gamma \in \Gamma$:

[1] $AP^+(\varphi)$, $AP^-(\varphi)$, $Reg^+(\varphi)$ and $Reg^-(\varphi)$ are as defined in Section 3.2.

1. if $\psi = a(x_1,...,x_n)$ or $\psi = \neg a(x_1,...,x_n)$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} \langle (\!|p,\psi|\!), \gamma\rangle \in \Delta'$;
2. if $\psi = \psi_1 \wedge \psi_2$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle (\!|p,\psi_1|\!), \gamma\rangle, \langle (\!|p,\psi_2|\!), \gamma\rangle] \in \Delta'$;
3. if $\psi = \psi_1 \vee \psi_2$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} \langle (\!|p,\psi_1|\!), \gamma\rangle \in \Delta'$ and $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} \langle (\!|p,\psi_2|\!), \gamma\rangle \in \Delta'$;
4. if $\psi = \exists x\, \psi_1$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{meet^x_{\{c\}}}{\hookrightarrow} \langle (\!|p,\psi_1|\!), \gamma\rangle \in \Delta'$, for every $c \in \mathcal{D}$;
5. if $\psi = \forall x\, \psi_1$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{meet^x_{\mathcal{D}}}{\hookrightarrow} [\langle (\!|p,\psi_1|\!), \gamma\rangle, \cdots, \langle (\!|p,\psi_1|\!), \gamma\rangle] \in \Delta'$, where $\langle (\!|p,\psi_1|\!), \gamma\rangle$ is repeated m times in $[\langle (\!|p,\psi_1|\!), \gamma\rangle, \cdots, \langle (\!|p,\psi_1|\!), \gamma\rangle]$, where m is the number of elements in $\mathcal{D}$;
6. if $\psi = EX\psi_1$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} \langle (\!|p',\psi_1|\!), \omega\rangle \in \Delta'$ for every $\langle p,\gamma\rangle \hookrightarrow \langle p',\omega\rangle \in \Delta$;
7. if $\psi = AX\psi_1$; $\langle (\!|p,\psi)|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle (\!|p_1,\psi_1|\!), \omega_1\rangle, \ldots, (\!|p_l,\psi_1|\!), \omega_l\rangle] \in \Delta'$ such that for every i, $1 \leq i \leq l$, $\langle p,\gamma\rangle \hookrightarrow \langle p_i,\omega_i\rangle \in \Delta$ and these transitions are all the transitions of Δ that have $\langle p,\gamma\rangle$ as left hand side;
8. if $\psi = E[\psi_1 U \psi_2]$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle (\!|p,\psi_1|\!), \gamma\rangle, \langle (\!|p',\psi|\!), \omega\rangle] \in \Delta'$ for every rule $\langle p,\gamma\rangle \hookrightarrow \langle p',\omega\rangle \in \Delta$, and $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} \langle (\!|p,\psi_2|\!), \gamma\rangle \in \Delta'$;
9. if $\psi = A[\psi_1 U \psi_2]$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle (\!|p,\psi_1|\!), \gamma\rangle, \langle (\!|p_1,\psi|\!), \omega_1\rangle, ..., \langle (\!|p_l,\psi|\!), \omega_l\rangle] \in \Delta'$ such that for every i, $1 \leq i \leq l$, $\langle p,\gamma\rangle \hookrightarrow \langle p_i,\omega_i\rangle \in \Delta$ and these transitions are all the transitions of Δ that have $\langle p,\gamma\rangle$ as left hand side, and $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} \langle (\!|p,\psi_2|\!), \gamma\rangle \in \Delta'$;
10. if $\psi = E[\psi_1 \tilde{U} \psi_2]$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle (\!|p,\psi_2|\!), \gamma\rangle, \langle (\!|p',\psi|\!), \omega\rangle] \in \Delta'$ for every $\langle p,\gamma\rangle \hookrightarrow \langle p',\omega\rangle \in \Delta$, and $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle (\!|p,\psi_2|\!), \gamma\rangle, \langle (\!|p,\psi_1|\!), \gamma\rangle] \in \Delta'$;
11. if $\psi = A[\psi_1 \tilde{U} \psi_2]$; $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle (\!|p,\psi_2|\!), \gamma\rangle, \langle (\!|p_1,\psi|\!), \omega_1\rangle, ..., \langle (\!|p_l,\psi|\!), \omega_l\rangle] \in \Delta'$ such that for every i, $1 \leq i \leq l$, $\langle p,\gamma\rangle \hookrightarrow \langle p_i,\omega_i\rangle \in \Delta$ and these transitions are all the transitions of Δ that have $\langle p,\gamma\rangle$ as left hand side, and $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle (\!|p,\psi_1|\!), \gamma\rangle, \langle (\!|p,\psi_2|\!), \gamma\rangle] \in \Delta'$;
12. if $\psi = e$: $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} \langle s_e, \gamma\rangle \in \Delta'$, where s_e is the initial state of M_e,
13. if $\psi = \neg e$: $\langle (\!|p,\psi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} \langle s_{\neg e}, \gamma\rangle \in \Delta'$, where $s_{\neg e}$ is the initial state of $M_{\neg e}$,
14. for every transition $q \xrightarrow{\alpha} \{q_1,...,q_n\}$ in $\mathcal{M}$; $\langle q,\gamma\rangle \overset{\Re}{\hookrightarrow} \{\langle q_1,\epsilon\rangle, ..., \langle q_n,\epsilon\rangle\} \in \Delta'$, where
    (a) $\Re = equal$ if $\alpha = \gamma$,
    (b) $\Re = join^x_\gamma$ if $\alpha = x \in \mathcal{X}$,
    (c) $\Re = join^{\neg x}_\gamma$ if $\alpha = \neg x$ and $x \in \mathcal{X}$,
15. for every $q \in \mathcal{F}$; $\langle q,\sharp\rangle \overset{equal}{\hookrightarrow} \langle q,\sharp\rangle \in \Delta'$.

Roughly speaking, $\mathcal{BP}_\varphi$ could be seen as the product of $\mathcal{P}$ and φ. $\mathcal{BP}_\varphi$ recognizes all the configurations $\langle [(\!|p,\psi|\!), B], \omega\rangle$ s.t. $\langle p,\omega\rangle$ satisfies ψ under B. Thus $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!|p,\psi|\!), B], \omega\rangle$ if and only if the configuration $\langle p,\omega\rangle$ satisfies ψ under B. The intuition behind each rule is explained in Appendix E. Here, due to lack of space, we only explain some of the rules.

If $\psi = a(x_1,...,x_n) \in AP^+(\varphi)$, then for every $\omega \in \Gamma^*$, $\langle p,\omega\rangle$ satisfies ψ under any environment B such that $p \in \lambda\Big(a(B(x_1),...,B(x_n))\Big)$. Thus, for such B's, $\mathcal{BP}_\varphi$ should have an accepting run from the configuration $\langle [(\!|p, a(x_1,...,x_n)|\!), B], \omega\rangle$. This is ensured by Item 1 that adds a loop in $\langle [(\!|p, a(x_1,...,x_n)|\!), B], \omega\rangle$ (since all accepting paths are infinite), and by the fact that the state $[(\!|p, a(x_1,...,x_n)|\!), B]$ is accepting thanks to F_1. Here the function is *equal* to ensure that the environment does not change while applying the rule.

If $\psi = \exists x\ \psi_1$, then for every $\omega \in \Gamma^*$, $B \in \mathcal{B}$, $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!|p,\psi|\!), B], \omega\rangle$ iff there exists $c \in \mathcal{D}$ such that $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!|p,\psi_1|\!), B[x \leftarrow c]], \omega\rangle$ which ensures that $\langle p,\omega\rangle$ satisfies ψ under the environment B iff $\langle p,\omega\rangle$ satisfies ψ_1 under $B[x \leftarrow c]$. This is expressed by Item 4 since $B \in meet^x_{\{c\}}(B[x \leftarrow c])$.

If $\psi = e$, then the SABPDS $\mathcal{BP}_\varphi$ accepts $\langle [(\!|p, \psi|\!), B], \omega \rangle$ iff $(\langle p, \omega \rangle, B) \in L(M_e)$. To check this, $\mathcal{BP}_\varphi$ first goes to state $[s_e, B]$ by Item 12, where s_e is the initial state of M_e, then it continues to check whether ω is accepted by M_e under the environment B. This is ensured by Items 14. Item 14 allows $\mathcal{BP}_\varphi$ to mimic a run of M_e on ω under the environment B: if $\mathcal{BP}_\varphi$ is in state $[q, B]$ and the topmost symbol of its stack is γ, then:

- Item 14(a) deals with the case where $q \xrightarrow{\gamma} \{q_1, ..., q_2\}$ is a transition in δ_e. In this case, $\mathcal{BP}_\varphi$ moves to the next states $[q_1, B], ..., [q_n, B]$ while popping γ from the stack. Popping γ allows $\mathcal{BP}_\varphi$ to check the rest of the word. The function *equal* guarantees that all the environments are the same.
- Item 14(b) deals with the case where $q \xrightarrow{x} \{q_1, ..., q_2\}$, $x \in \mathcal{X}$ is a transition in δ_e. In this case, $\mathcal{BP}_\varphi$ can continue to mimic a run of M_e under the environment B only if $B(x) = \gamma$. If this holds, $\mathcal{BP}_\varphi$ moves to the next states $[q_1, B], ..., [q_n, B]$ and pops γ from the stack, which allows $\mathcal{BP}_\varphi$ to check the rest content of the stack. The function $join^x_\gamma$ ensures that all the environments are the same and the value of $B(x)$ is γ.
- Item 14(c) deals with the case where $q \xrightarrow{\neg x} \{q_1, ..., q_2\}$ is a transition in δ_e. In this situation, $\mathcal{BP}_\varphi$ can continue to mimic a run of M_e under the environment B only if $B(x) \neq \gamma$. If this holds, $\mathcal{BP}_\varphi$ moves to the next states $[q_1, B], ..., [q_n, B]$ and pops γ from the stack. The function $join^{\neg x}_\gamma$ ensures that all the environments are the same and the value of $B(x)$ is different from γ.

Thus, $(\langle p, \omega \rangle, B) \in L(M_e)$ iff M_e reaches final states $f_1, ..., f_n$ of M_e after reading the word ω, i.e., iff $\mathcal{BP}_\varphi$ reaches a set of states $[f_1, B], ..., [f_n, B]$ with an empty stack (a stack containing only the bottom stack symbol $\sharp$). This is why F_4 is a set of accepting states. Moreover, since all the accepting paths are infinite, Item 15 adds a loop on every configuration $\langle [f, B], \sharp \rangle$ where f is a final state of M and $\sharp$ is the stack symbol (this makes the paths of $\mathcal{BP}_\varphi$ that reach a state $\langle [f, B], \sharp \rangle$ accepting). Formally, we can show:

Theorem 3. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a function $\lambda : \mathrm{AP}_{\mathcal{D}} \longrightarrow 2^P$, a SCTPL formula φ, and a configuration $\langle p, \omega \rangle$ of $\mathcal{P}$, we have: for every $B \in \mathcal{B}$, $\langle p, \omega \rangle \models^B_\lambda \varphi$ iff $\mathcal{BP}_\varphi$ has an accepting run from the configuration $\langle [(\!|p, \varphi|\!), B], \omega \rangle$.*

4.4 Computing $\mathcal{L}(\mathcal{BP})$

Let $\mathcal{BP} = (P, \Gamma, \Delta, F)$ be a SABPDS. In this section, we give an algorithm to compute a SAMA that recognizes $\mathcal{L}(\mathcal{BP})$. First, we characterize the set of configurations from which the SABPDS has an accepting run. Then, we show how to compute this set.

Characterizing $\mathcal{L}(\mathcal{BP})$. Let $Y_{\mathcal{BP}} = \bigcap_{i \geq 0} X_i$ where $X_0 = (P \times \mathcal{B}) \times \Gamma^*$ and for every $i \geq 0$, $X_{i+1} = Pre^+(X_i \cap F \times \Gamma^*)$, where $F \times \Gamma^*$ stands for $\{\langle [p, B], \omega \rangle \in (P \times \mathcal{B}) \times \Gamma^* \mid \exists [p, \beta] \in F \ s.t. \ B \in \beta \}$. We can show that:

Proposition 2. *Given a SABPDS $\mathcal{BP} = (P, \Gamma, \Delta, F)$, $\mathcal{L}(\mathcal{BP}) = Y_{\mathcal{BP}}$.*

Computing $Y_{\mathcal{BP}}$. Our goal is to compute $Y_{\mathcal{BP}} = \bigcap_{i \geq 0} X_i$. We provide a symbolic algorithm that computes this set. Our procedure is an extension of the procedure given in [11, 24] that computes an AMA recognizing the language of an ABPDS. We show that $Y_{\mathcal{BP}}$ can be represented by a SAMA $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$ whose set of states Q is a subset of

$(P \times 2^{\mathcal{B}}) \times \mathbb{N} \cup \{q_f\}$, where q_f is a special state that corresponds to the unique final state ($Q_f = \{q_f\}$). For every $[p,\beta] \in P \times 2^{\mathcal{B}}$ and $i \in \mathbb{N}$, let $[p,\beta]^i$ denote $([p,\beta], i)$. To compute $Y_{\mathcal{BP}}$, we iteratively compute a SAMA A_i using states of the form $[p,\beta]^i$ during the step i. Moreover, We extend the function $\mathfrak{R} : (\mathcal{B})^n \longrightarrow 2^{\mathcal{B}}$ to $\mathfrak{R} : (2^{\mathcal{B}})^n \longrightarrow 2^{\mathcal{B}}$ as follows: $\mathfrak{R}(\beta_1, ..., \beta_n) = \{B \in \mathcal{B} \mid B \in \mathfrak{R}\{B_1, ..., B_n\}$ s.t. for every $1 \leq i \leq n : B_i \in \beta_i\}$; and we define to functions π^{-1} and π^i as follows: For every $S \subseteq Q$,

$$\pi^{-1}(S) = \begin{cases} \{q^i \mid q^{i+1} \in S\} \cup \{q_f\} \text{ if } q_f \in S \text{ or } \exists q^1 \in S, \\ \{q^i \mid q^{i+1} \in S\} \qquad \text{else.} \end{cases}$$

$$\pi^i(S) = \{q^i \mid \exists 1 \leq j \leq i \; s.t. \; q^j \in S\} \cup \{q_f \mid q_f \in S\}.$$

```
Input:      A SABPDS $\mathcal{BP} = (P, \Gamma, \Delta, F, \mathcal{X}, \mathcal{D})$.
Output:     A SAMA $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$ that recognizes $Y_{\mathcal{BP}}$, where $Q \subseteq (P \times 2^{\mathcal{B}}) \times \mathbb{N} \cup \{q_f\}$, $Q_f = \{q_f\}$.
Initially:  Let $i = 0, \delta = \{(q_f, \gamma, \{q_f\}) \mid$ for every $\gamma \in \Gamma\}$, and for $[p,\beta] \in F : [p,\beta]^0 = q_f$.
1.          Repeat (we call this loop $loop_1$)
2.            $i := i + 1$;
3.            Add in $\delta$ a new transition $[p,\beta' \cap \beta]^i \xrightarrow{\epsilon} [p,\beta']^{i-1}$, for every $[p,\beta] \in F, [p,\beta']^{i-1} \xrightarrow{\gamma} Q \in \delta$;
4.            Repeat (we call this loop $loop_2$)
5.              For every $\langle p, \gamma \rangle \overset{\mathfrak{R}}{\hookrightarrow} [\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle]$ in $\Delta$,
6.              and every case where $[p_k, \beta_k]^i \xrightarrow{\omega_k}_{\delta} Q_k$, for all $1 \leq k \leq n$;
7.                Add a new rule $[p,\beta]^i \xrightarrow{\gamma} \bigcup_{k=1}^{n} Q_k$ in $\delta$ where $\beta = \mathfrak{R}(\beta_1, ..., \beta_n)$;
8.            Until No new transition rule can be added.
9.            Remove from $\delta$ the transition rules added by line 3;
10.           Replace in $\delta$ every transition rule $[p,\beta]^i \xrightarrow{\gamma} R$ by $[p,\beta]^i \xrightarrow{\gamma} \pi^i(R)$, for every $\gamma \in \Gamma, R \subseteq Q$;
11.         Until $i > 1$ and $\forall [p,\beta] \in P \times 2^{\mathcal{B}}, \gamma \in \Gamma, R \subseteq (P \times 2^{\mathcal{B}}) \times \{i\} \cup \{q_f\} : [p,\beta]^i \xrightarrow{\gamma} R \in \delta$ iff $[p,\beta]^{i-1} \xrightarrow{\gamma} \pi^{-1}(R) \in \delta$
```

Table 1. Algorithm 1: Computation of $Y_{\mathcal{BP}}$

Algorithm 1 computes a SAMA $\mathcal{A}$ recognizing $Y_{\mathcal{BP}}$. To understand the idea behind this algorithm, let A_0 be the automaton obtained after the initialization step and A_i be the automaton obtained at step i (a step starts at Line 2) for every $i \geq 1$. Each state $[p,\beta]^i$ represents the state $[p,\beta]$ at step i, i.e., A_i recognizes a configuration $\langle [p, B], \omega \rangle$ iff there exists $\beta \subseteq \mathcal{B}$ s.t. $[p,\beta]^i \xrightarrow{\omega}_{\delta} q_f$ and $B \in \beta$. It is clear that A_0 recognizes $X_0 \cap F \times \Gamma^*$. Suppose the algorithm is at the beginning of the i^{th} iteration ($loop_1$). Line 3 adds the ϵ-transition $[p,\beta' \cap \beta]^i \xrightarrow{\epsilon} [p,\beta']^{i-1}$, for every $[p,\beta] \in F s.t. [p,\beta']^{i-1} \xrightarrow{\gamma} Q \in \delta$. After this step, we obtain $L(A_{i-1}) \cap F \times \Gamma^*$. $loop_2$ (Lines 4–8) is the saturation procedure that computes the Pre^* of $L(A_{i-1}) \cap F \times \Gamma^*$. Line 9 removes the ϵ-transition added by Line 3. After Line 9, the automaton A_i recognizes $Pre^+(L(A_{i-1}) \cap F \times \Gamma^*)$. Thus, in case of termination, the algorithm produces $Y_{\mathcal{BP}}$. The substitution at Line 10 is needed to guarantee the termination of the algorithm. We show that: (a sketch of the proof is given in the appendix)

Theorem 4. Algorithm 1 *always terminates and produces $Y_{\mathcal{BP}}$.*

Thus, we get: (the complexity is discussed in the appendix)

Theorem 5. *Let $\mathcal{BP} = (P, \Gamma, \Delta, F)$ be a SABPDS, then we can compute a SAMA $\mathcal{A}$ that recognizes $\mathcal{L}(\mathcal{BP})$ in $O\Big(|P|^2 \cdot 2^{2|\mathcal{B}|} \cdot |\Gamma| \cdot |\Delta| \cdot 2^{5|P| \cdot 2^{|\mathcal{B}|}}\Big)$ time.*

Remark 4. Note that another way to compute $\mathcal{L}(\mathcal{BP})$ is to apply Lemma 1 and produce an equivalent ABPDS $\mathcal{BP}'$ that simulates $\mathcal{BP}$, and then apply the algorithm of [24] to compute an AMA that recognizes $\mathcal{L}(\mathcal{BP}')$. The complexity of such a procedure would be $O(|P|^2 \cdot |\Delta| \cdot |\mathcal{B}|^{h+3} \cdot |\Gamma| \cdot 2^{5|P|\cdot|\mathcal{B}|})$, where h is the maximum of the widths of the transition rules in Δ . This worst case complexity is better than the complexity of **Algorithm 1**. However, in practice, in the symbolic case (for SABPDS), the sets of environments β's can be compactly represented using BDDs for example, whereas in the explicit case (for ABPDS), all the environments B's have to be considered. Thus, **Algorithm 1** will behave better in practice. This is confirmed by the experiments we run where, in the majority of cases, **Algorithm 1** terminates in few seconds, whereas if we compute an equivalent ABPDS and apply the algorithm of [24], we run out of memory. These experimental results are summarized in Section 5 and Table 4 in the appendix.

4.5 SCTPL model-checking for PDSs

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a labeling function λ, and a SCTPL formula φ, thanks to Theorems 3 and 5, and due to the fact that $\mathcal{BP}_\varphi$ has $O(|P| \cdot |\varphi| + k)$ states and $O((|P| \cdot |\Gamma| + |\Delta|) \cdot |\varphi| + d)$ transitions, where k and d are the number of states and the number of transitions of the union $\mathcal{M}$ of the Variable Automata involved in φ; we get the following:

Corollary 1. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a SCTPL formula φ and a labeling function λ, we can effectively compute a SAMA $\mathcal{A}$ in time $O\Big((|P||\varphi| + k)^2 \cdot 2^{2|\mathcal{B}|} \cdot |\Gamma| \cdot ((|P||\Gamma| + |\Delta|)|\varphi| + d) \cdot 2^{5(|P||\varphi|+k)\cdot 2^{|\mathcal{B}|}}\Big)$, where k is the number of states of $\mathcal{M}$ and d is the number of transition rules of $\mathcal{M}$ such that for every configuration $\langle p, \omega \rangle$ of $\mathcal{P}$:*

1. *$\langle p, \omega \rangle \models_\lambda \varphi$ iff there exists a $B \in \mathcal{B}$ s.t. $\mathcal{A}$ recognizes $\langle [(\!|p, \varphi|\!), B], \omega \rangle$.*
2. *for every $B \in \mathcal{B}$: $\langle p, \omega \rangle \models_\lambda^B \varphi$ iff $\mathcal{A}$ recognizes $\langle [(\!|p, \varphi|\!), B], \omega \rangle$.*

Thus, thanks to this corollary and to Proposition 1, it follows that it is possible to determine whether a PDS configuration satisfies a SCTPL formula:

Corollary 2. *It is possible to decide whether a PDS configuration satisfies a SCTPL formula.*

Remark 5. As described in Remark 2, we can transform every SCTPL formula ψ to an equivalent CTL with regular valuations formula ψ' such that $|\psi'| = O(|\psi| \cdot |\mathcal{D}|^g)$ where g is the number of subformulas of ψ in the form of $\forall x\ \varphi$ or $\exists x\ \varphi$. Every regular variable expression in ψ will generate $|\mathcal{D}|^{|\mathcal{X}|}$ "standard" regular expressions over Γ in ψ'. Thus, the number of states $|\mathcal{S}'|$ and the number of transition rules $|\mathcal{T}'|$ of the finite automata corresponding to the regular expressions in ψ' will be $|\mathcal{D}|^{|\mathcal{X}|} \cdot k$ and $|\mathcal{D}|^{|\mathcal{X}|} \cdot d$, respectively. Then, applying [24], we can construct an AMA recognizing all the configurations which satisfy ψ' in time $O(|P|^3 \cdot |\Gamma|^2 \cdot |\psi'|^3 \cdot |\mathcal{S}'|^2 \cdot |\Delta| \cdot |\mathcal{T}'| \cdot 2^{5(|P||\psi'|+|\mathcal{S}'|)})$, i.e., $O(|P|^3 \cdot |\Gamma|^2 \cdot |\psi|^3 \cdot |\mathcal{D}|^{3g} \cdot |\mathcal{B}|^3 \cdot k^2 \cdot |\Delta| \cdot d \cdot 2^{5(|P|(|\psi|\cdot|\mathcal{D}|^g)+|\mathcal{B}|\cdot k)})$, where $|\mathcal{B}| = |\mathcal{D}|^{|\mathcal{X}|}$.

This theoretical complexity is better than the complexity of Corollary 1 obtained using our SCTPL model-checker. However, in practice, thanks to the compact representation of the sets of environments β's using BDDs, model-checking SCTPL using our symbolic techniques behaves much better than reducing SCTPL to CTL with regular valuations and then applying [24]. Indeed, the experiments we run show that in most of the cases, our symbolic algorithm for SCTPL model-checking terminates in few seconds, whereas translating the SCTPL formula to CTL with regular valuations and then applying [24] would run out of memory. These experimental results are summarized in Section 5 and Table 4 in the appendix.

Examples	$\vert P\vert$	Our techniques		SABPDS→ABPDS		SCTPL→CTLr		Result
		Time(s)	Mem(Mb)	Time(s)	Mem(Mb)	Time(s)	Mem(Mb)	
Klez.a	42	1.62	10.8	-	MemOut	-	MemOut	Y
Klez.b	45	1.55	10.8	-	MemOut	-	MemOut	Y
Klez.c	41	1.27	8.9	-	MemOut	-	MemOut	Y
Klez.d	51	1.47	10.3	-	MemOut	-	MemOut	Y
Klez.e	52	0.77	7.0	-	MemOut	-	MemOut	Y
Klez.f	50	0.76	7.0	-	MemOut	-	MemOut	Y
Klez.g	47	0.75	7.0	-	MemOut	-	MemOut	Y
Klez.i	49	0.74	7.0	-	MemOut	-	MemOut	Y
Klez.j	55	0.74	7.0	-	MemOut	-	MemOut	Y
Mydoom.c	210	145.20	322.8	-	MemOut	-	MemOut	Y
Mydoom.e	288	123.22	267.5	-	MemOut	-	MemOut	Y
Mydoom.g	256	117.50	256.7	-	MemOut	-	MemOut	Y
Predec.j	25	0.23	0.81	-	MemOut	56.14	36.16	Y
Netsky.a	69	2.73	14.5	-	MemOut	-	MemOut	Y
Akez	42	0.22	0.3	-	MemOut	0.44	2.49	Y
Netsky.b	80	2.73	14.5	-	MemOut	-	MemOut	Y
Netsky.c	78	2.73	14.5	-	MemOut	-	MemOut	Y
Netsky.d	72	2.73	14.5	-	MemOut	-	MemOut	Y
Alcaul.h	48	0.83	0.9	-	MemOut	1.14	6.88	Y
Uedit32	180	92.58	100.94	-	MemOut	-	MemOut	N
Alcaul.l	52	0.30	0.7	-	MemOut	0.86	3.96	Y
Cygwin32	212	23.72	123.31	-	MemOut	-	MemOut	N
cmd.exe	202	1.44	25.52	-	MemOut	-	MemOut	N
Alcaul.o	68	0.20	0.6	-	MemOut	0.83	3.37	Y
Mydoor.ar	256	113.2	227.4	-	MemOut	-	MemOut	Y
Adson.1559	52	0.22	2.1	-	MemOut	-	MemOut	Y
Adson.1651	54	0.23	2.1	-	MemOut	-	MemOut	Y
Adson.1703	55	0.25	2.1	-	MemOut	-	MemOut	Y
Adson.1734	54	0.31	2.6	-	MemOut	-	MemOut	Y
Alcaul.d	62	0.20	0.8	-	MemOut	47.70	51	Y
Alcaul.i	88	4.38	0.28	-	MemOut	159.88	169.64	Y
Alcaul.j	79	0.30	2.1	-	MemOut	218.25	198.71	Y
Oroch.3982	89	3.70	7.72	-	MemOut	-	MemOut	Y
KME	145	999.31	20.04	-	MemOut	-	MemOut	Y
Anar.a	41	1.16	1.60	885.33	343.24	54.92	34.12	Y
Anar.b	47	1.49	1.60	891.42	348.54	56.14	36.16	Y
Atak.b	126	762.34	18.15	-	MemOut	-	MemOut	Y
Alcaul.c	33	0.12	0.3	-	MemOut	0.41	2.19	Y
Bagle.d	88	652.23	16.96	-	MemOut	-	MemOut	Y
Alcaul.f	52	0.09	0.3	-	MemOut	0.53	2.23	Y
Alcaul.b	50	0.06	0.2	-	MemOut	0.28	1.18	Y
Alcaul.e	49	0.49	0.9	-	MemOut	1.03	5.28	Y
Alcaul.g	53	0.31	0.7	-	MemOut	0.97	4.45	Y
Evol.a	102	9.58	3.22	-	MemOut	-	MemOut	Y
Alcaul.k	52	0.26	0.6	-	MemOut	0.76	3.65	Y
Alcaul.m	53	0.20	0.6	-	MemOut	0.88	3.37	Y
Alcaul.n	34	0.12	0.3	-	MemOut	0.44	2.28	Y
Klinge	78	237.50	4.49	-	MemOut	0.83	3.37	Y
Atak.f	220	23.4	139.1	-	MemOut	-	MemOut	Y
Mydoor.ay	328	124.2	232.5	-	MemOut	-	MemOut	Y

Table 2. Detection of real malwares

5 Experiments

We implemented our techniques in a tool for malware detection. We use IDAPro [3] as disassembler. We use BDDs to represent sets of environments. We carried out different experiments. We obtained interesting results. In particular, our tool was able to detect several viruses taken from [17]. Our results are reported in Table 2.

Column $|P|$ gives the number of control locations of the PDS model. Every program is checked against several malicious behaviors. A program is declared as a potential virus if it satisfies one of the specifications. **Column** *time(s)* and *mem(Mb)* give the time (in seconds) and the memory (in Mb). The last **Column** *result* is *Y* is the program contains the malicious behaviors given in **Column** *Formula*, and *N* if not. We also compared our techniques against translating SABPDS to ABPDS (**Columns** "SABPDS→ABPDS"), or translating SCTPL to CTL with regular valuations (**Columns** "SCTPL→CTLr"). We were able to detect all the viruses that we considered, whereas applying the translation from SABPDS to ABPDS or from SCTPL to CTL with regular valuations would run out of memory in most of the cases, and thus cannot detect the viruses. Our tool was also able to deduce that some benign programs are not viruses. E.g. we tried the following benign programs: *Uedit32*, a fragment of Ultra Edit Text Editor software by IDM Computer Solutions; *Cygwin32* a fragment of the Setup software of Cygwin, a Linux-like environment for Windows. *cmd.exe* is the Microsoft-supplied command-line interpreter.

Moreover, we run several experiments to check how robust are our techniques in virus detection in case the virus writers use obfuscation techniques. To this aim, we considered some of the viruses of Table 2, and we added several obfuscations manually such as: instruction reordering (reordering the instructions inside the code and using jump instructions so that the control flow is not changed), dead code insertion, register renaming, splitting the code into several procedures, adding useless stack operations, etc. We tested 5 variants for each type of obfuscation of the viruses Mydoom.g, Netsky.a, Bagle.d, Adson.1734 and Akez. The results are reported in Table 3. Our techniques were able to detect all these variations, whereas the three well known and widely used free antiviruses *Avira* [2], *Qihoo 360* [4] and *Avast* [1] were not able to detect several of these virus variations.

Obfuscation	**Our techniques detection rate**	**Avira antivirus detection rate**	**Qihoo 360 antivirus detection rate**	**Avast antivirus detection rate**
nop-insertion	**100**%	65%	55%	60%
code-reordering	**100**%	40%	35%	45%
register-renaming	**100**%	25%	25%	30%
stack-operation	**100**%	20%	25%	20%
procedure-split	**100**%	5%	5%	5%

Table 3. Detection of obfuscated Viruses

References

1. Avast antivirus, free version. `http://www.avast.com`.
2. Avira antivirus, free version. `http://www.avira.com`.
3. IDA Pro. `http://www.hex-rays.com/idapro/`.
4. Qihoo 360 antivirus. `http://www.360.cn`.
5. G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In *CC*, 2005.
6. G. Balakrishnan, T. W. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. H. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with codesurfer/x86 and wpds++. In *CAV*, 2005.
7. J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *SREIS*, 2001.
8. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, 5:263–270, 2009.
9. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *CONCUR'97*. LNCS 1243, 1997.
10. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3), 1992.
11. T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *ICALP*, 2002.
12. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, 2003.
13. M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ISEC*, 2008.
14. M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, 2005.
15. J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput.*, 186(2), 2003.
16. J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *CAV*, 2001.
17. V. Heavens. http://vx.netlux.org.
18. A. Holzer, J. Kinder, and H. Veith. Using verification technology to specify and detect malware. In *EUROCAST*, 2007.
19. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *DIMVA*, 2005.
20. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing*, 7(4), 2010.
21. A. Lakhotia, D. R. Boccardo, A. Singh, and A. Manacero. Context-sensitive analysis of obfuscated x86 executables. In *PEPM*, 2010.
22. A. Lakhotia, E. U. Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Trans. Software Eng.*, 31(11), 2005.
23. P. K. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *IAW*, 2003.
24. F. Song and T. Touili. Efficient CTL model-checking for pushdown systems. In *CONCUR*, 2011.
25. D. Suwimonteerabuth, S. Schwoon, and J. Esparza. Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In *ATVA*, 2006.

A Experiments: Symbolic vs. explicit

As described previously, our approach consists in computing a SABPDS from the PDS and the SCTPL formula, and then applying **Algorithm 1** to compute the set of configurations from which the SABPDS has an accepting run, i.e., that satisfy the SCTPL formula. As explained in Remarks 4 and 5, this can be done differently in two ways: (1) either translate the SABPDS into an equivalent ABPDS and then apply the algorithm of [24] to compute the set of configurations that it accepts; (2) or translate the SCTPL formula into an equivalent CTL with regular valuations formula, and then apply an existing algorithm for model-checking PDSs against CTL with regular valuations (such as the one given in [24]). In order to show that our approach is much better than these two solutions, we run several experiments that compares the three approaches. Our experiments are applied to random PDSs. The results are summarized in Table 4. **Column** *PDS* $|P|+|\Gamma|+|\Delta|$ gives the number of control locations, the number of stack alphabet and the number of transitions of the PDS. **Column** *SCTPL size* denotes the size of the considered SCTPL formula. **Columns** $|\mathcal{X}|$ and $|\mathcal{D}|$ denote the number of variables and the size of the domain. The **Columns** "Our techniques" describe the results obtained using our techniques. The **Columns** "SABPDS→ABPDS" describe the results obtained if we translate the SABPDS to an equivalent ABPDS and then apply the algorithm of [24]. The last **Columns** "SCTPL→CTLr" describe the results obtained if the SCTPL formula is translated into a CTL with regular valuations formula. $|\Delta_s|$ and $|\delta_s|$ denote the number of transitions of the SABPDSs and the SAMAs computed by **Algorithm 1**. $|\Delta_1|$ denotes the number of transitions of the ABPDSs corresponding to the SABPDSs. $|\delta_1|$ gives the number of transitions of the AMAs computed using the algorithm of [24]. **Column** *time(s)* and *mem(kb)* give the time (in seconds) and the memory (in kilobytes). **Memout** means "memory out" (the memory limit is 650Mb). The results described in Table 4 show that our techniques behave much better than the two other techniques. In most of the cases, our techniques terminate in few seconds and using less memory, whereas the two other approaches run out of memory.

B From binary code to Pushdown Systems

We represent a binary code program by a set of control flow graphs (CFGs), one CFG for each procedure. These CFGs are over-approximations of the concrete program. The nodes of a CFG correspond to the program locations, and its edges are annotated with assembly instructions (e.g. *mov eax,0*). Several tools allow to extract a set of CFGs from a binary code, such as IDAPro [3], CodeSurfer/x86 [5], etc. We can use these tools to extract CFGs from binary code. Some of these tools involve efficient static analysis techniques that allow to compute over-approximations of the sets of numeric values and addresses that are involved in every control point of the program. In particular, they provide informations on the possible values of l in instructions of the form n_1 : *jmp l* or n_1 : *call l*. Thus, we suppose that in the CFGs, these instructions are represented by edges of the form $n_1 \xrightarrow{jmp\ n} n$ and $n_1 \xrightarrow{call\ n} n_2$ for all the possible values n of l (these values are computed by the CFG-extractor tool). Moreover, we suppose that pushes and pops can be done only using *push*, *pop*, *call*, and *return* operations, not by manipulating the stack pointer.

Translation:

Given a set of CFGs S, we define a corresponding PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ such that: Γ is the set of symbols α such that there exists in S an edge of the form $n_1 \xrightarrow{push\ \alpha} n_2$ or

PDS	SCTPL			Our techniques				SABPDS→ABPDS				SCTPL→CTLr	
$\lvert P\rvert+\lvert \Gamma\rvert+\lvert \Delta\rvert$	size	$\lvert \mathcal{X}\rvert$	$\lvert \mathcal{D}\rvert$	SABPDS	SAMA	Time	Mem	ABPDS	AMA	Time	Mem	Time	Mem
				$\lvert \Delta_s\rvert$	$\lvert \delta_s\rvert$	(s)	(Kb)	$\lvert \Delta_1\rvert$	$\lvert \delta_1\rvert$	(s)	(kb)	(s)	(kb)
1+2+1	3	2	4	18	6	0.02	27	432	12	0.03	54	0.03	41
1+2+1	4	2	4	20	8	0.02	27	464	20	0.02	56	0.03	42
1+2+1	3	2	5	22	6	0.00	27	1072	14	0.02	94	0.03	47
1+2+1	4	2	5	25	8	0.03	28	1147	28	0.03	102	0.02	48
4+5+3	6	3	11	693	30	0.02	76	**5257k**	120	40.67	**329k**	0.33	1236
4+5+3	3	1	13	301	59	0.00	55	2225	169	0.03	190	0.02	140
4+5+3	6	3	13	813	30	0.03	84	-	-	-	**MemOut**	0.56	1767
4+5+3	5	2	9	393	26	0.02	56	23448	66	0.14	1505	0.05	115
4+5+3	6	3	9	573	30	0.02	67	**939k**	92	6.75	**59k**	0.19	845
4+4+5	4	2	10	357	66	0.05	61	96597	335	1.05	6147	0.20	968
4+4+5	6	3	10	521	60	0.02	69	**8529k**	1077	65.28	**634k**	0.09	525
4+4+5	4	3	7	373	48	0.02	57	**939k**	92	6.8	**59k**	0.06	281
4+4+5	6	3	8	425	60	0.02	63	**1895k**	609	13.53	**119k**	0.06	346
4+4+5	6	3	9	473	60	0.02	66	**4213k**	819	29.81	**264k**	0.12	425
4+4+5	6	3	11	569	60	0.03	72	-	-	-	**MemOut**	0.16	622
4+4+5	6	3	12	617	60	0.05	75	-	-	-	**MemOut**	0.17	724
4+4+5	6	3	20	1001	60	0.05	99	-	-	-	**MemOut**	0.97	2096
12 +12+6	11	1	16	1752	187	0.06	197.60	-	-	-	**MemOut**	0.36	1094
12 +12+6	13	3	16	1896	187	0.11	776.91	-	-	-	**MemOut**	197.94	**27.14k**
12 +12+6	15	5	24	5928	340	0.27	1.60k	-	-	-	**MemOut**	-	**MemOut**
25 +35+6	15	5	34	16878	691	1.12	5.72k	-	-	-	**MemOut**	-	**MemOut**
34 +65+6	15	5	50	32808	967	4.33	14.99k	-	-	-	**MemOut**	-	**MemOut**
42+96+6	15	5	58	46641	1284	8.05	24.11k	-	-	-	**MemOut**	-	**MemOut**
50+124+6	15	5	70	66360	1555	17.84	40.23k	-	-	-	**MemOut**	-	**MemOut**
66+169+7	15	5	71	103437	2334	28.81	63.60k	-	-	-	**MemOut**	-	**MemOut**
75+215+7	11	4	14	19051	2493	0.66	4.87k	-	-	-	**MemOut**	-	**MemOut**
75+215+7	15	5	26	46696	2697	2.02	13.14k	-	-	-	**MemOut**	-	**MemOut**
75+215+7	15	5	59	98699	2724	17.20	51.76k	-	-	-	**MemOut**	-	**MemOut**
75+215+7	17	7	59	160649	2738	18.34	55.57k	-	-	-	**MemOut**	-	**MemOut**
75+215+7	17	7	99	265677	2771	100.00	140.02k	-	-	-	**MemOut**	-	**MemOut**
75+215+7	17	7	139	370698	2799	341.91	262.91k	-	-	-	**MemOut**	-	**MemOut**
75+215+7	17	7	174	462601	2837	880.61	401.99k	-	-	-	**MemOut**	-	**MemOut**

Table 4. Our techniques vs. Explicit techniques.

$n_1 \xrightarrow{call\ proc} \alpha$. P is $\Gamma \cup N$, where N is the set of nodes of S; and Δ contains transition rules that mimic the behaviors of the program's instructions. Let i be an instruction from control point n_1 to control point n_2:

- if i is of the form $n_1 \xrightarrow{push\ \alpha} n_2$, it is translated into a set of push rules $\langle n_1, \gamma\rangle \hookrightarrow \langle n_2, \alpha\gamma\rangle$, for every $\gamma \in \Gamma$;
- if i is of the form $n_1 \xrightarrow{pop\ \alpha} n_2$, it is translated into a set of pop rules $\langle n_1, \gamma\rangle \hookrightarrow \langle n_2, \epsilon\rangle$, for every $\gamma \in \Gamma$, where ϵ is the empty word;
- if i is a call instruction $n_1 \xrightarrow{call\ proc} n_2$, it is translated into a set of push rules $\langle n_1, \gamma\rangle \hookrightarrow \langle e_{proc}, n_2\, \gamma\rangle$, for every $\gamma \in \Gamma$. These rules move the control point to the entry point e_{proc} of the procedure *proc* and pushes the return address n_2 of the callee onto the stack. This is how an assembly program behaves when it executes a call.
- if i is a return instruction $n_1 \xrightarrow{ret} n_2$, it is translated into a set of pop rules $\langle n_1, \gamma\rangle \hookrightarrow \langle \gamma, \epsilon\rangle$, for every $\gamma \in \Gamma$. These rules remove the topmost symbol γ from the stack, and moves the PDS's control point to γ, i.e., to the return address. This is how an assembly program behaves when it executes a return.

– if i is any other instruction $n_1 \xrightarrow{\textit{other instruction}} n_2$, it is translated into a set of rules $\langle n_1, \gamma\rangle \hookrightarrow \langle n_2, \gamma\rangle$, for every $\gamma \in \Gamma$. These rules move the PDS's control point from n_1 to n_2 without changing the stack.

Note that in our modeling, the PDS control locations correspond to the program's control points, and the PDS stack mimics the program's execution stack. The above transition rules allow the PDS to mimic the behavior of the program's stack. This is different from standard program translations to PDSs where the control points of the program are stored in the stack [16, 6]. These standard translations assume that the program follows a standard compilation model, where the return addresses are never modified. We do not make such assumptions since behaviors where the return addresses are modified can occur in malicious code. We only make the assumption that pushes and pops can be done only using *push*, *pop*, *call*, and *return* operations, not by manipulating the stack pointer.

Example: The fragment of code of Figure 1(b) can be encoded by the PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ such that: $P = \{l'_1, l'_2, l'_3, l'_4, l'_5, l'_6, g_0\}$ where g_0 is the entry point of the function *GetModuleHandleA* and l'_6 is the location just after l'_5. The stack alphabet is $\Gamma = \{eax, ebx, l'_6\}$. The transition rules Δ are shown in Figure 2.

$$\boxed{\forall \gamma \in \{eax, ebx, l'_6\}}$$

$$\boxed{\begin{array}{l}\langle l'_1, \gamma\rangle \hookrightarrow \langle l'_2, \gamma\rangle\\ \langle l'_2, \gamma\rangle \hookrightarrow \langle l'_3, eax\ \gamma\rangle\\ \langle l'_3, \gamma\rangle \hookrightarrow \langle l'_4, ebx\ \gamma\rangle\\ \langle l'_4, \gamma\rangle \hookrightarrow \langle l'_5, \epsilon\rangle\\ \langle l'_5, \gamma\rangle \hookrightarrow \langle g_0, l'_6\gamma\rangle\end{array}}$$

Fig. 2. Δ

C An example illustrating SCTPL

Example: Consider the fragment of Figure 1(b), and the SCTPL formula ψ described in the introduction by the formula (2). In this example, we have:

$$\boxed{\begin{array}{l}\lambda(mov(eax, 0)) = \{l'_1\}\\ \lambda(push(eax)) = \{l'_2\}\\ \lambda(push(ebx)) = \{l'_3\}\\ \lambda(pop(ebx)) = \{l'_4\}\\ \lambda(call(GetModuleHandleA)) = \{l'_5\}\end{array}}$$

Fig. 3. λ.

– $\mathcal{X} = \{r_1, r_2, r_3\}$ is the set of variables appearing in ψ,
– $\mathcal{R} = \{r_1\Gamma^*\}$ is the set of regular variable expressions in ψ,
– $AP = \{mov, push, pop, call\}$ is the set of atomic propositions corresponding to the instructions of the program,
– $AP_{\mathcal{X}} = \{mov(r_1, 0), mov(r_1, r_2), push(r_1), pop(r_3), call(GetModuleHandleA)\}$ is the set of predicates that appear in the formula ψ,
– $\mathcal{D} = \{eax, ebx, 0, GetModuleHandleA, l'_6\}$ is defined such that $\Gamma \subseteq \mathcal{D}$, and the set of instructions of the program are in $AP_{\mathcal{D}}$,
– $AP_{\mathcal{D}} = \{mov(eax, 0), push(eax), push(ebx), pop(ebx), call(GetModuleHandleA)\}$ is the set of labels of the program's instructions,
– The labeling function λ is described in Figure 3.

Consider the PDS of Figure 2 that describes this fragment of code. Any configuration $\langle l'_1, w\rangle$, $w \in \Gamma^*$ satisfies the subformula φ:

$$\varphi = \mathbf{EF}\Big(mov(r_1, 0) \wedge \mathbf{EX}\ \mathbf{E}[\neg\exists r_2\ mov(r_1, r_2)\mathbf{U}(push(r_1) \wedge \mathbf{EX}\ \mathbf{E}[\neg\Big(push(r_1)\vee$$
$$(\exists r_3(pop(r_3) \wedge r_1\Gamma^*))\Big)\mathbf{U}(call(GetModuleHandleA) \wedge r_1\Gamma^*)])]\Big)$$

under all the environments B s.t. $B(r_1) = eax$. Thus, since $\psi = \exists\ r_1\ \varphi$, we get that any configuration $\langle l'_1, w\rangle$, $w \in \Gamma^*$ satisfies the specification ψ under every environment $B' \in \mathcal{B}$.

```
...
lea eax, [ebp + ExistingFileName]
push eax
push 0
call ds : GetModuleFileNameA
...
lea eax, [ebp + ExistingFileName]
push eax
call ds : CopyFileA
...
```

(a)

$$\begin{array}{ll}
1. & \psi_{ew} = \exists m\Big(\exists r_0\Big(\\
2. & \mathbf{EF}\big(lea(r_0, m) \wedge \mathbf{EX}\ \mathbf{E}\Big[\neg\exists v(mov(r_0, v) \vee lea(r_0, v))\mathbf{U}\big(push(r_0) \\
3. & \quad \wedge\, \mathbf{EX}\ \mathbf{E}[\neg(push(r_0) \vee \exists v(pop(v) \wedge r_0\Gamma^*))\mathbf{U}(call(GetModuleFileNameA) \wedge 0\ r_0\Gamma^* \\
4. & \qquad \wedge\, \exists r_1\big(\mathbf{EF}(lea(r_1, m) \wedge \mathbf{EX}\ \mathbf{E}[\neg\exists v(mov(r_1, v) \vee lea(r_1, v))\mathbf{U}(push(r_1) \\
5. & \qquad\quad \wedge\, \mathbf{EX}\ \mathbf{E}[\neg(push(r_1) \vee \exists v(pop(v) \wedge r_1\Gamma^*))\mathbf{U}call(CopyFileA) \wedge r_1\Gamma^*])]))))]\big)\Big]\big)\Big)\Big)
\end{array}$$

(b)

Fig. 5. (a) Email worm (b) Specification of Email worm

D Modeling malicious behaviors using SCTPL

In this section, we show some examples that illustrate how SCTPL can be used to precisely specify malicious behaviors. We needed stack predicates to express most of the specifications. Except the first specification given using a CTPL formula, all the other malicious behaviors described in this section need to use predicates over the stack. Thus, SCTPL is necessary to specify these behaviors, CTPL is not sufficient.

Kernel32.dll base address viruses. Many of Windows viruses use an API to achieve their malicious tasks. The Kernel32.dll file includes several API functions that can be used by the viruses. In order to use these functions, the viruses have to find the entry addresses of these API functions. To do this, they need to determine the Kernel32.dll entry point. They determine first the Kernel32.dll PE header in memory and use this information to locate Kernel32.dll export section and find the entry addresses of the API functions. For this, the virus looks first for the DOS header (the first word of the DOS header is *5A4Dh* in hex (*MZ* in ascii)); and then looks for the PE header (the first two words of the PE header is *4550h* in hex (*PE*00 in ascii)). Figure 4 presents a disassembled code fragment performing this malicious behavior. This can be specified in SCTPL as follows:

```
l1 : cmp [eax], 5A4Dh
jnz l2
...
cmp [ebx], 4550h
jz l3
l2 : ...
jmp l1
l3
```

Fig. 4. Virus.

$$\psi_{wv} = EG\big(EF(\exists r_1\ cmp(r_1, 5A4Dh) \wedge EF\ \exists r_2\ cmp(r_2, 4550h))\big).$$

This SCTPL formula expresses that the program has a loop such that there are two variables r_1 and r_2 such that first, r_1 is compared to *5A4Dh*, and then r_2 is compared to *4550h*. Note that this formula can detect all the class of viruses that have such behavior.

Email worms. The typical behavior of an email worm can be summarized as follows: the worm will first call the API *GetModuleFileNameA* in order to get the name of its executable. For this, the worm needs to call this function with 0 and m as parameters (m corresponds to the address of a memory location), i.e., with $0m$ on the top of the stack since parameters to a function in assembly are passed through the stack. *GetModuleFileNameA* will then write the name of the worm executable on the address m. Then, the worm will copy its file (whose name is at the address m) to other locations using the function *CopyFileA*. It needs to call *CopyFileA* with m as parameter, i.e., with m on the top of the stack. Figure 5(a) shows a disassembled fragment of a code corresponding to this typical behavior. This behavior can be expressed by the SCTPL formula of Figure 5(b). In this formula, Line 2 expresses that there exists a register r_0 such that the address of the memory location m is assigned to r_0, and such that the value of r_0 does not change until it is pushed onto the stack (subformula $\neg\exists v(mov(r_0,v)\vee lea(r_0,v))\ U push(r_0)$). Line 3 guarantees that r_0 is not pushed nor popped from the stack until *GetModuleFileNameA* is called, and $0r_0$ is on the top of the stack (the predicate $0r_0\Gamma^*$ ensures this). This guarantees that when *GetModuleFileNameA* is called, r_0 still contains the address of m. Thus, the name of the worm file returned by *GetModuleFileNameA* will be put at the address m. Line 4 is similar to Line 2. It expresses that there exists a register r_1 such that the address of the memory location m is assigned to r_1, and such that the value of r_1 does not change until it is pushed onto the stack. This guarantees that when r_1 is pushed to the stack, it contains the address of m. Line 5 expresses that r_1 is not pushed nor popped from the stack until *CopyFileA* is called, and r_1 is on the top of the stack (the predicate $r_1\Gamma^*$ ensures this). This guarantees that when *CopyFileA* is called, the value of r_1 is still m. Thus, *CopyFileA* will copy the file whose name is at the address m. Note that we need predicates over the stack to express in a precise manner this specification.

Obfuscated calls. Virus writers try to obfuscate their code by e.g. hiding the calls to the operating system. For example, a *call* instruction can be replaced by pushes and jumps. Figure 6 shows two equivalent fragments achieving a "call" instruction. Figure 6(a) shows a normal call/ret where the function f consists just of a *return* instruction. When control point f is reached, the *return* instruction moves the control point to l_1 which is the return address of the call instruction (at l_0). As shown in Figure 6(b), the *call* can be equivalently substituted by two other instructions, where *push* l'_2 pushes the return address l'_2 onto the stack, and *jmp f* moves the control point to the entry point of f. These instructions do exactly the same thing than the *call* instruction. When reaching the control point f, the *ret* instruction will pop the stack and thus, move the control point to l'_2. Such obfuscated calls can be described by the following SCTPL formula:

l_0 : call f	l'_0 : push l'_2
l_1 : ...	l'_1 : jmp f
	l'_2 : ...
f: ret	
	f: ret
(a)	(b)

Fig. 6. (a) Normal call. (b) Obfusated call

$$\psi_{oc} = \exists\, addr\ \mathbf{E}[\neg(\exists\, proc\ call(proc) \wedge\ \mathbf{EX}\ addr\Gamma^*)\ \mathbf{U}\ (ret \wedge addr\Gamma^*)]$$

The subformula ($\exists\, proc\ call(proc) \wedge \mathbf{EX}\ addr\Gamma^*$) means that there exists a procedure call having *addr* as return address, since when a procedure call is made, the program will push its corresponding return address *addr* to the stack, and thus, at the next step, we will have *addr* on the top of the stack (i.e., $addr\Gamma^*$). The subformula ($ret \wedge addr\Gamma^*$) expresses that we have a return instruction with *addr* on the top of the stack, i.e., a return instruction that will return to *addr*. Thus the formula ψ_{oc} expresses that there exists a return address *addr* such that there exists a path where there is no call to a procedure *proc* having *addr* as return address until a return instruction with *addr* as return address occurs. This formula can then detect a return that does not correspond to a call.

Obfuscated returns. Virus writers usually obfuscate the returns of their calls in order to make it difficult to manually or automatically analyse their code. Benign programs move the control point to the return address using the *ret* instruction. Viruses may replace the *ret* instruction by other equivalent instructions such as *pop eax*, *jmp l*, etc. E.g., the program in Figure 7 is a disassembled fragment from the virus Klinge that pops the return address *00401028* from the stack. This phenomenon can be detected by the following specification:

```
00401023: call 004011CE
00401028: ...
    ...
004011CE:  ...
    ...
0040121A: pop eax
```

Fig. 7. Fragment of the Virus Klinge

$$\psi_{or} = \mathbf{AG}\Big(\forall proc \forall addr((call(proc) \wedge \mathbf{AX}\, addr\Gamma^*) \Longrightarrow \mathbf{AF}(ret \wedge addr\Gamma^*))\Big).$$

ψ_{or} expresses that for every procedure *proc*, if *proc* is called with *addr* as the return address of the caller, then there exists a *ret* instruction which will return to *addr*. Indeed, since when an assembly program runs, if an instruction *call proc* is executed, then the return address *addr* of the caller is pushed onto the stack. Thus, in the subformula $call(proc) \wedge \mathbf{AX}\, addr\Gamma^*$, *addr* refers to the return address of the call, because this subformula expresses that in all the immediate successors of the call, *addr* is on the top of the stack. Moreover, $ret \wedge addr\Gamma^*$ means that when the return is executed, then the return address *addr* should be on the top of the stack.[2]

Appending viruses. An appending virus is a virus that inserts a copy of its malicious code at the end of the target file. To do this, the virus has to first calculate its real absolute address in the memory, because the real OFFSET of the virus' variables depends on the size of the infected file. To achieve this, the viruses have to call the routine in Figure 8 (this code is a fragment of the virus Alcaul.b). The instruction *call* l_2 will push the return address l_2 onto the stack. Then, the *pop* instruction will put the value of this address into the register *eax*. In this way, the virus can get its real absolute address in the memory. This malicious behavior can be detected using the specification ψ_{or}, since there does not exist any *return* instruction corresponding to the *call* instruction.

```
l1 : call l2
l2 : pop eax
    ...
```

Fig. 8.

E Intuition behind the rules of $\mathcal{BP}_\varphi$

As said previously, $\mathcal{BP}_\varphi$ could be seen as the product of $\mathcal{P}$ and φ. $\mathcal{BP}_\varphi$ recognizes all the configurations $\langle [(\!|p, \psi|\!), B], \omega\rangle$ s.t. $\langle p, \omega\rangle$ satisfies ψ under B. Thus $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!|p, \psi|\!), B], \omega\rangle$ if and only if the configuration $\langle p, \omega\rangle$ satisfies ψ under B. The intuition behind each rule is explained as follows.

If $\psi = a(x_1, ..., x_n) \in AP^+(\varphi)$, then for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle$ satisfies ψ under any environment B such that $p \in \lambda\Big(a(B(x_1), ..., B(x_n))\Big)$. Thus, for such B's, $\mathcal{BP}_\varphi$ should have an accepting run from the configuration $\langle [(\!|p, a(x_1, ..., x_n)|\!), B], \omega\rangle$. This is ensured by Item 1 that adds a loop in $\langle [(\!|p, a(x_1, ..., x_n)|\!), B], \omega\rangle$ (since all accepting paths are infinite), and by the fact that the state $[(\!|p, a(x_1, ..., x_n)|\!), B]$ is accepting thanks to F_1. Here the function is *id* to ensure that the environment does not change while applying the rule.

If $\psi = \neg a(x_1, ..., x_n) \in AP^-(\varphi)$, then for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle$ satisfies ψ under any environment B such that $p \notin \lambda\Big(a(B(x_1), ..., B(x_n))\Big)$. Thus, for such B's, $\mathcal{BP}_\varphi$ should have an accepting run from the configuration $\langle [(\!|p, \neg a(x_1, ..., x_n)|\!), B], \omega\rangle$. This is ensured by Item 1

[2] Note that for the case of a procedure that has a possibly infinite loop, this specification can detect a suspected malware. This formula can be changed slightly to avoid this. We do not present this here for the sake of presentation.

that adds a loop in $\langle[(\!|p, \neg a(x_1, ..., x_n)|\!), B], \omega\rangle$ (all accepting paths are infinite). The definition of F_2 guarantees that the state $[(\!|p, \neg a(x_1, ..., x_n)|\!), B]$ is accepting.

If $\psi = \psi_1 \wedge \psi_2$, Item 2 ensures that for every $\omega \in \Gamma^*$, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \psi_1 \wedge \psi_2|\!), B], \omega\rangle$ iff it has an accepting run from $\langle[(\!|p, \psi_1|\!), B], \omega\rangle$ and from $\langle[(\!|p, \psi_2|\!), B], \omega\rangle$. This means that $\langle p, \omega\rangle$ satisfies ψ under B iff $\langle p, \omega\rangle$ satisfies ψ_1 and ψ_2 under B. The function *equal* ensures that the environment B is the same for these three states. The intuition behind Item 3 is similar.

If $\psi = \exists x\ \psi_1$, then for every $\omega \in \Gamma^*$, $B \in \mathcal{B}$, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \psi|\!), B], \omega\rangle$ iff there exists $c \in \mathcal{D}$ such that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \psi_1|\!), B[x \leftarrow c]], \omega\rangle$ which ensures that $\langle p, \omega\rangle$ satisfies ψ under the environment B iff $\langle p, \omega\rangle$ satisfies ψ_1 under $B[x \leftarrow c]$. This is expressed by Item 4 since $B \in meet^x_{\{c\}}(B[x \leftarrow c])$.

If $\psi = \forall x\ \psi_1$, then for every $\omega \in \Gamma^*$, $B \in \mathcal{B}$, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \psi|\!), B], \omega\rangle$ iff for every $c \in \mathcal{D}$, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \psi_1|\!), B[x \leftarrow c]], \omega\rangle$ which ensures that $\langle p, \omega\rangle$ satisfies ψ under the environment B iff $\langle p, \omega\rangle$ satisfies ψ_1 under $B[x \leftarrow c]$ for every $c \in \mathcal{D}$. This is guaranteed by Item 5 and its corresponding function $meet^x_{\mathcal{D}}$ since if $\mathcal{D} = \{c_1, \ldots, c_m\}$, then $B \in meet^x_{\mathcal{D}}(B[x \leftarrow c_1], \ldots, B[x \leftarrow c_m])$.

If $\psi = EX\psi_1$, then for every $p \in P$, $\omega \in \Gamma^*$ and $B \in \mathcal{B}$, $\langle p, \omega\rangle$ satisfies ψ under B iff there exists an immediate successor $\langle p', \omega'\rangle$ of $\langle p, \omega\rangle$ such that $\langle p', \omega'\rangle$ satisfies ψ_1 under B. Thus, $\mathcal{BP}_\varphi$ should have an accepting run from $\langle[(\!|p, \psi|\!), B], \omega\rangle$ iff it has an accepting run from $\langle[(\!|p', \psi_1|\!), B], \omega'\rangle$. This is expressed by Item 6 where the function *id* guarantees that the environment remains the same.

If $\psi = AX\psi_1$, then for every $p \in P$, $\omega \in \Gamma^*$ and $B \in \mathcal{B}$, $\langle p, \omega\rangle$ satisfies ψ under B iff $\langle p_j, \omega_j\rangle$ satisfies ψ_1 under B for every immediate successor $\langle p_j, \omega_j\rangle$ of $\langle p, \omega\rangle$. This means that $\mathcal{BP}_\varphi$ should have an accepting run from $\langle[(\!|p, \psi|\!), B], \omega\rangle$ iff it has an accepting run from every configuration $\langle[(\!|p_j, \psi_1|\!), B], \omega_j\rangle$. Item 7 expresses this. The function *equal* makes sure that all these environments are the same.

If $\psi = E[\psi_1 U \psi_2]$, then for every $p \in P$, $\omega \in \Gamma^*$ and $B \in \mathcal{B}$, $\langle p, \omega\rangle$ satisfies ψ under B iff either it satisfies ψ_2 under B, or it satisfies ψ_1 under B and it has an immediate successor that satisfies ψ under B. This is expressed by Item 8. The case $\psi = A[\psi_1 U \psi_2]$ is analogous.

If $\psi = E[\psi_1 \tilde{U} \psi_2]$, then for every $p \in P$, $\omega \in \Gamma^*$, and $B \in \mathcal{B}$, $\langle p, \omega\rangle$ satisfies ψ under B iff it satisfies ψ_2 under B, and either it satisfies also ψ_1 under B, or it has an immediate successor that satisfies ψ under B. This is expressed by Item 10. This ensures that either ψ_2 holds always, or until both ψ_1 and ψ_2 hold. F_3 ensures that $[(\!|p, \psi|\!), B]$ is accepting for every $B \in \mathcal{B}$, i.e., that a path where ψ_2 always hold is accepting. The case where $\psi = A[\psi_1 \tilde{U} \psi_2]$ is similar.

If $\psi = e$, then the SABPDS $\mathcal{BP}_\varphi$ accepts $\langle[(\!|p, \psi|\!), B], \omega\rangle$ iff $(\langle p, \omega\rangle, B) \in L(M_e)$. To check this, $\mathcal{BP}_\varphi$ first goes to state $[s_e, B]$ by Item 12, where s_e is the initial state of M_e, then it continues to check whether ω is accepted by M_e under the environment B. This is ensured by Items 14. Item 14 allows $\mathcal{BP}_\varphi$ to mimic a run of M_e on ω under the environment B: if $\mathcal{BP}_\varphi$ is in state $[q, B]$ and the topmost of its stack is γ, then:

- Item 14(a) deals with the case where $q \xrightarrow{\gamma} \{q_1, ..., q_2\}$ is a transition in δ_e. In this case, $\mathcal{BP}_\varphi$ moves to the next states $[q_1, B], ..., [q_n, B]$ while popping γ from the stack. Popping γ allows $\mathcal{BP}_\varphi$ to check the rest of the word. The function *equal* guarantees that all the environments are the same.
- Item 14(b) deals with the case where $q \xrightarrow{x} \{q_1, ..., q_2\}$, $x \in \mathcal{X}$ is a transition in δ_e. In this case, $\mathcal{BP}_\varphi$ can continue to mimic a run of M_e under the environment B only if $B(x) = \gamma$. If this holds, $\mathcal{BP}_\varphi$ moves to the next states $[q_1, B], ..., [q_n, B]$ and pops γ from

the stack, which allows $\mathcal{BP}_\varphi$ to check the rest content of the stack. The function $join^x_\gamma$ ensures that all the environments are the same and the value of $B(x)$ is γ.

- Item 14(c) deals with the case where $q \xrightarrow{\neg x} \{q_1, ..., q_2\}$ is a transition in δ_e. In this situation, $\mathcal{BP}_\varphi$ can continue to mimic a run of M_e under the environment B only if $B(x) = \neg\gamma$. If this holds, $\mathcal{BP}_\varphi$ moves to the next states $[q_1, B], ..., [q_n, B]$ and pops γ from the stack. The function $join^{\neg x}_\gamma$ ensures that all the environments are the same and the value of $B(x)$ is different from γ.

Thus, $(\langle p, \omega\rangle, B) \in L(M_e)$ iff M_e reaches final states $f_1, ..., f_n$ of M_e after reading the word w, i.e., iff $\mathcal{BP}_\varphi$ reaches a set of states $[f_1, B], ..., [f_n, B]$ with an empty stack (a stack containing only the bottom stack symbol $\sharp$). This is why F_4 is a set of accepting states. Moreover, since all the accepting paths are infinite, Item 15 adds a loop on every configuration $\langle[f, B], \sharp\rangle$ where f is a final state of M and $\sharp$ is the stack symbol (this makes the paths of $\mathcal{BP}_\varphi$ that reach a state $\langle[f, B], \sharp\rangle$ accepting).

The case where $\psi = \neg e$ is similar to the previous case.

F Proofs of Section 4.4

Proposition 2. Given a SABPDS $\mathcal{BP} = (P, \Gamma, \Delta, F)$, $\mathcal{L}(\mathcal{BP}) = Y_{\mathcal{BP}}$.
Proof (Sketch): The proof follows the lines of [24], where it was shown that for an ABPDS $\mathcal{BP}' = (P, \Gamma, \Delta, F)$, $\mathcal{L}(\mathcal{BP}')$ is equal to $\bigcap_{i\geq 0} Z_i$, where $Z_0 = P \times \Gamma^*$ and $Z_{i+1} = Pre^+(Z_i \cap F \times \Gamma^*)$. Here, $X_0 = (P \times \mathcal{B}) \times \Gamma^*$ since configurations of the SABPDS $\mathcal{BP}$ are in $P \times \mathcal{B} \times \Gamma^*$. □

Theorem 4. Algorithm 1 always terminates and produces $Y_{\mathcal{BP}}$.
Proof (Sketch): The proof follows the lines of the proof of [24]. Indeed, our algorithm follows the idea of the algorithm that computes an AMA recognizing the language of an ABPDS given in [24]. The main differences are:

1. We use states of the form $[p, \beta]$ instead of p for every $p \in P$, since we now deal with SABPDS. A symbolic state $[p, \beta] \in P \times 2^{\mathcal{B}}$ denotes a set of states $[p, B]$ for every environment $B \in \beta$ which records the valuation of the variables $\mathcal{X}$;
2. To compute the Pre^* of $L(A_{i-1}) \cap F \times \Gamma^*$, instead of using the following saturation procedure given in [9] that computes the Pre^* for alternating pushdown systems:
If $\langle p, \gamma\rangle \hookrightarrow \{\langle p_1, \omega_1\rangle, ..., \langle p_n, \omega_n\rangle\}$ and $p^i_k \xrightarrow{\omega_k}_\delta Q_k$ for all $1 \leq k \leq n$, add a transition $p^i \xrightarrow{\gamma} \bigcup^n_{k=1} Q_k$.
We use the following saturation procedure: If $\langle p, \gamma\rangle \overset{\mathfrak{R}}{\hookrightarrow} [\langle p_1, \omega_1\rangle, ..., \langle p_n, \omega_n\rangle]$ and $[p_k, \beta_k]^i \xrightarrow{\omega_k}_\delta Q_k$ for all $1 \leq k \leq n$, add a transition $[p, \beta]^i \xrightarrow{\gamma} \bigcup^n_{k=1} Q_k$, where $\beta = \mathfrak{R}(\beta_1, ..., \beta_n)$.
Indeed, intuitively, if e.g. $\langle[p, B], \gamma\omega\rangle$ is an immediate predecessor of $\{\langle[p_1, B_1], \omega_1\omega\rangle, \langle[p_2, B_2], \omega_2\omega\rangle\}$ by the transition rule $\langle p, \gamma\rangle \overset{\mathfrak{R}}{\hookrightarrow} [\langle p_1, \omega_1\rangle, \langle p_2, \omega_2\rangle]$, and $\langle[p_1, B_1], \omega_1\omega\rangle$ and $\langle[p_2, B_2], \omega_2\omega\rangle$ are in $L(A_{i-1}) \cap F \times \Gamma^*$, then, necessarily, $B \in \mathfrak{R}(B_1, B_2)$ and there exist $\beta_1, \beta_2 \subseteq \mathcal{B}$ and $S_1, S_2 \subseteq Q$ s.t. $B_1 \in \beta_1$, $B_2 \in \beta_2$, $[p_1, \beta_1] \xrightarrow{\omega_1}_\delta S_1 \xrightarrow{\omega}_\delta q_f$ and $[p_2, \beta_2] \xrightarrow{\omega_2}_\delta S_2 \xrightarrow{\omega}_\delta q_f$. Lines 4 – 8 add the new transition $[p, \mathfrak{R}(\beta_1, \beta_2)] \xrightarrow{\gamma}_\delta S_1 \cup S_2$. This allows to accept the configuration $\langle[p, B], \gamma\omega\rangle$ using the run $[p, \mathfrak{R}(\beta_1, \beta_2)] \xrightarrow{\gamma}_\delta S_1 \cup S_2 \xrightarrow{\omega}_\delta q_f$.

3. In Line 3, instead of adding a new ϵ transition rule $p^i \xrightarrow{\epsilon} p^{i-1}$ for every $p \in F$, we add a new ϵ transition rule $[p,\beta \cap \beta']^i \xrightarrow{\epsilon} [p,\beta']^{i-1}$ for every $[p,\beta] \in F$ such that $[p,\beta']^{i-1} \xrightarrow{\gamma} Q \in \delta$. Since this step is used to compute $L(A_{i-1}) \cap F \times \Gamma^*$ and $F \times \Gamma^*$ stands for $\{\langle [p,B], \omega\rangle \in (P \times \mathcal{B}) \times \Gamma^* \mid \exists [p,\beta] \in F \ s.t. \ B \in \beta\}$, it is not correct if we only add the ϵ-transition $[p,\beta]^i \xrightarrow{\epsilon} [p,\beta]^{i-1}$, for every $[p,\beta] \in F$. Indeed, it is possible that A_{i-1} recognizes some configurations $\{\langle [p,B], \gamma\omega\rangle \mid B \in \beta'\} \supset \{\langle [p,B], \gamma\omega\rangle \mid B \in \beta\}$ by a path $[p,\beta']^{i-1} \xrightarrow{\gamma\omega} q_f$ where $[p,\beta] \in F$ whereas $[p,\beta'] \notin F$.

□

F.1 Complexity of Theorem 4

Given an alternating pushdown system with P as set of control states and an AMA A with n states and having P as the set of initial states, [25] provides a way to efficiently implement the saturation procedure of [9] that computes the Pre^* of A in time $O(n \cdot |\Delta| \cdot 2^{2n})$. We can show that because of the substitution at Line 10, at each step i, **Algorithm 1** only needs to consider states of the form $[p,\beta]^i$ and $[p,\beta]^{i-1}$ in addition to q_f. Since the arbitrary functions $\mathfrak{R}$ can generate all the possible $\beta \subseteq \mathcal{B}$, the number of states at each step i should be $2|P| \cdot 2^{|\mathcal{B}|}$. $loop_2$ in **Algorithm 1** can be seen as an extension of the saturation procedure of [25]. Thus, by adapting the complexity analysis of [25], we can show that $loop_2$ needs $O(|P| \cdot 2^{|\mathcal{B}|} \cdot |\Delta| \cdot 2^{4|P| \cdot 2^{|\mathcal{B}|}})$ time. The substitution (Line 10) and termination condition (Line 11) can be done in time $O(|P| \cdot 2^{|\mathcal{B}|} \cdot |\Gamma| \cdot 2^{2|P| \cdot 2^{|\mathcal{B}|}})$ and $O(|P| \cdot 2^{|\mathcal{B}|} \cdot |\Gamma| \cdot 2^{|P| \cdot 2^{|\mathcal{B}|}})$, respectively. Putting all these estimations together, the global complexity of **Algorithm 1** is $O\Big(|P|^2 \cdot 2^{2|\mathcal{B}|} \cdot |\Gamma| \cdot |\Delta| \cdot 2^{5|P| \cdot 2^{|\mathcal{B}|}}\Big)$.