Int J Softw Tools Technol Transfer (2014) 16:147–173
DOI 10.1007/s10009-013-0290-1

TACAS 2012

Pushdown model checking for malware detection

Fu Song · **Tayssir Touili**

Published online: 30 September 2013

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 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 study, 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 stack computation tree predicate logic (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 applied it to detect several viruses. Our results are encouraging.

Keywords Pushdown Systems · Model Checking · CTL · Malware Detection

Work partially funded by ANR Grant ANR-08-SEGI-006, Shanghai Knowledge Service Platform for Trustworthy Internet of Things No. ZF1213 and NSFC Project No.91118007. Civil Aerospace Project 125, NSFC Project No. 61021004.

F. Song (✉)
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, 200062 Shanghai, People's Republic of China
e-mail: fsong@sei.ecnu.edu.cn

T. Touili
Liafa, CNRS and Université Paris Diderot, Case 7014, 75205 Paris Cedex 13, France
e-mail: touili@liafa.univ-paris-diderot.fr

1 Introduction

The number of malwares that produced incidents in 2010 is more than 1.5 billion [17]. A malware may bring serious damage, e.g., the worm MyDoom slowed down global internet access by 10 % in 2004 [13]. Thus, it is crucial to have efficient up-to-date virus detectors. Existing antivirus systems use various detection techniques to identify viruses such as (1) code emulation where the virus is executed in a virtual environment to get detected; or (2) signature detection, where a signature is a pattern of program code that characterizes the virus. A file is declared as a virus if it contains a sequence of binary code instructions that matches one of the known signatures. Each virus variant has its corresponding signature. These techniques have some limitations. Indeed, emulation-based techniques can only check the program's behavior in a limited time interval. They cannot check what happens after the timeout. Thus, they might miss the viral behavior if it occurs after this time interval. As for signature-based systems, it is very easy to virus developers to get around them. It suffices to apply obfuscation techniques to change the structure of the code while keeping the same functionality, so that the new version does not match the known signatures. Obfuscation techniques can consist in inserting dead code, substituting instructions by equivalent ones, etc. Virus writers update their viruses frequently to make them undetectable by these antivirus systems.

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 study to use *model checking* for virus detection. Model checking has already been used for virus detection in [5,10,12,20–22,27]. 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 described in [25], 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 [25]. Therefore, it is important to have analysis techniques that can deal with the program stack.

We propose in this study 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 considering 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 [4,16]. 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, i.e., the data in the stack cannot be changed via direct memory access.

The logic SCTPL that we introduce is an extension of the CTPL logic that allows using the predicates over the stack. CTPL was introduced in [20–22]. 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 $x_1, \ldots, x_n$ 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

```
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

be expressed in CTL as a large formula enumerating all the possible registers:

$$\begin{aligned}&\mathbf{EF}\ \big(mov(eax, data) \wedge \mathbf{AF}\ push(eax)\big)\vee\\&\mathbf{EF}\ \big(mov(ebx, data) \wedge \mathbf{AF}\ push(ebx)\big)\vee\\&\mathbf{EF}\ \big(mov(ecx, data) \wedge \mathbf{AF}\ push(ecx)\big) \vee \ldots\end{aligned}$$

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 $\exists r\ \mathbf{EF}\ \big(mov(r, data) \wedge \mathbf{AF}\ push(r)\big)$ which expresses in a succinct way that there exists a *register r* such that the above holds. References [20–22] 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 Fig. 1a. It corresponds to a critical fragment of the Email-worm Avron [18] that shows the typical behavior of an email worm: it calls an API function *GetModuleHandleA* with 0 as its parameter. This allows getting 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:

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

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 Fig. 1a. However, a worm writer can easily use some obfuscation techniques 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 Fig. 1b. By doing so, this fragment keeps the same malicious behavior as that of the fragment in Fig. 1a. 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 to escape a given CTPL formula.

To overcome this problem, we introduce the SCTPL 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 existentially and universally quantified). Using our new logic SCTPL, the malicious behavior of Fig. 1a, b can be specified as follows:

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

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 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 Fig. 1, because it allows specifying the content of the stack when *GetModuleHandleA* is called. Note that it is important to use pushdown systems as model 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 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 propose an algorithm to solve this emptiness problem.
4. We implemented our techniques in a tool that we successfully applied to detect several viruses.

This paper is the full version of [29].

Outline. We give our translation from binary programs to PDSs in Sect. 2. In Sect. 3, we introduce our SCTPL and show how it can be used to precisely specify malicious behavior. Our SCTPL model-checking algorithm for pushdown systems is given in Sect. 4. The experiments we made for malware detection are reported in Sect. 5. Section 6 describes the related work.

2 Binary code modeling

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 extracting a set of CFGs from a binary code, such as IDA Pro [19], CodeSurfer/x86 [3], Jakstab [23], BAP [8], etc. We can use these tools to extract CFGs from binary code. Some of these tools involve efficient static analysis techniques that allow computing 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.

2.1 Formal model: pushdown systems

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

The reachability relation $\Longrightarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is the reflexive and transitive closure of the successor relation $\rightsquigarrow_{\mathcal{P}}$. Formally $\Longrightarrow_{\mathcal{P}}$ is defined as follows: (1) $c \Longrightarrow_{\mathcal{P}} c$ for every $c \in P \times \Gamma^*$, (2) if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then

$\langle p, \gamma\omega' \rangle \Longrightarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$, (3) if $c \Longrightarrow_{\mathcal{P}} c''$ and $c'' \Longrightarrow_{\mathcal{P}} c'$, then $c \Longrightarrow_{\mathcal{P}} c'$.

2.2 From binary code to pushdown systems

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 $n_1 \xrightarrow{call\ proc} \alpha$. 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 $n_1 \xrightarrow{other\ instruction} n_2$, 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 [4,16]. 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, i.e., the data in the stack cannot be changed via direct memory access.

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

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

Fig. 2 The transition rules Δ

```
l1:mov ebx 1
l2:call proc
l3:push ebx
l4:call GetModuleHandleA
```

(a)

```
l'1:mov ebx 0
l'2:call proc
l'3:push ebx
l'4:call GetModuleHandleA
```

(b)

Fig. 3 **a** A fragment of a benign program and **b** a fragment of a malware

Example: The fragment of code of Fig. 1b 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 Fig. 2.

Remark 1 In our binary code modeling, we push the names of the registers onto the stack. Instead, one can push the real values of the registers onto stack. This way, the behavior described in Sect. 1 can be expressed in a more precise way by the SCTPL formula $\psi' = \mathbf{EF}call(GetModuleHandleA) \wedge 0\Gamma^*$. ψ' expresses that there exists a path where the function *GetModuleHandleA* is called when 0 is on the top of the stack, i.e., 0 is the first parameter of *GetModuleHandleA*. This approach is more precise if there is a tool that can provide the real values of the registers and addresses. However, in practice, such a precise tool does not exist. All the existing tools will provide approximated values of the registers and addresses. Thus, in some situations, our approach consisting in pushing the names of the registers onto the stack works better. For example, let us consider the programs shown in Fig. 3a, b. Figure 3a is a fragment of a benign program calling *GetModuleHandleA* with 1 as its parameter, while Fig. 3b is a fragment of a malware calling *GetModuleHandleA* with 0 as its parameter. Then, ψ given in Eq. (2) can classify the malware and benign programs in Fig. 3a, b. However, if we rely on a tool to provide the values of the registers and addresses, if this tool is unable to provide the exact value of the register *ebx* at the control points l_3 and l'_3, respectively, and if the tool uses over-approximations, then both programs in Fig. 3a, b might be identified as malwares. While if the tool uses under-approximations, both programs in Fig. 3a, b might be identified as benign programs.

3 Malicious behavior specification

In this section, we introduce the stack computation tree predicate logic (SCTPL), and show how it can be used to specify malicious behavior.

3.1 Environments, predicates, and regular expressions

From now on, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, \ldots\}$ 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 $x \in \mathcal{X}$ 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 $AP = \{a, b, c, \ldots\}$ be a finite set of atomic propositions, $AP_{\mathcal{X}}$ be a finite set of atomic predicates of the form $b(\alpha_1, \ldots, \alpha_m)$ such that $b \in AP$ and $\alpha_i \in \mathcal{X} \cup \mathcal{D}$ for every i, $1 \le i \le m$, and $AP_{\mathcal{D}}$ be a finite set of atomic predicates of the form $b(\alpha_1, \ldots, \alpha_m)$ such that $b \in AP$ and $\alpha_i \in \mathcal{D}$ for every i, $1 \le i \le m$.

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 $\{(\langle p, \gamma \rangle, B) \mid p \in P, B \in \mathcal{B}\}$,
- $L(e_1 + e_2) = L(e_1) \cup L(e_2)$,
- $L(e_1 \cdot e_2) = \{(\langle p, \omega_1\omega_2 \rangle, B) \mid (\langle p, \omega_1 \rangle, B) \in L(e_1)$ and $(\langle p, \omega_2 \rangle, B) \in L(e_2)\}$,
- $L(e^*) = \{(\langle p, \omega \rangle, B) \mid \forall B \in \mathcal{B}\ s.t.\ \omega \in \{u \in \Gamma^* \mid (\langle p, u \rangle, B) \in L(e)\}^*\}$.

For example, $(\langle p, \gamma_1\gamma_1\gamma_2 \rangle, B)$ 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 expressing the 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 **R** 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: $\mathbf{EF}\psi = \mathbf{E}[true\mathbf{U}\psi]$, $\mathbf{AF}\psi = \mathbf{A}[true\mathbf{U}\psi]$, $\mathbf{EG}\psi = \mathbf{E}[false\mathbf{R}\psi]$ and $\mathbf{AG}\psi = \mathbf{A}[false\mathbf{R}\psi]$.

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

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

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

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:

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

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

Fig. 4 The labeling function λ

– $c \models^B_\lambda \mathbf{E}[\psi_1 \mathbf{R} \psi_2]$ iff there exists a path $\pi = c_0, c_1, \ldots,$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\forall i \geq 0$ s.t. $c_i \not\models^B_\lambda \psi_2$, $\exists 0 \leq j < i$ s.t. $c_j \models^B_\lambda \psi_1$.

Intuitively, $c \models^B_\lambda \psi$ holds iff the configuration c satisfies the formula ψ under the environment B. Note that a path π satisfies $\psi_1 \mathbf{R} \psi_2$ iff either ψ_2 holds everywhere in π or the first occurrence in the path where ψ_2 does not hold must be preceded by a position where ψ_1 holds.

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

– $\mathcal{X} = \{r_1, r_2, r_3\}$ is the set of variables that appear 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 Fig. 4.

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

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

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 $B' \in \mathcal{B}$.

Remark 2 CTPL [22] 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 expressing the predicates over the content of the stack using regular languages.

Remark 3 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.

3.3 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, where CTPL is not sufficient.

3.3.1 Kernel32.dll base address viruses

Many 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 ($PE00$ in ascii)). Figure 5 presents a disassembled code fragment performing

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

Fig. 5 Virus

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

(a)

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. $\wedge\, \mathbf{EX}\ \mathbf{E}[\neg(push(r_0) \vee \exists v(pop(v) \wedge r_0\Gamma^*))\mathbf{U}(call(GetModuleFileNameA) \wedge 0\ r_0\Gamma^*$
4. $\wedge\, \exists r_1 \Big(\mathbf{EF}(lea(r_1, m) \wedge \mathbf{EX}\ \mathbf{E}[\neg\exists v(mov(r_1, v) \vee lea(r_1, v))\mathbf{U}(push(r_1)$
5. $\wedge\, \mathbf{EX}\ \mathbf{E}[\neg(push(r_1) \vee \exists v(pop(v) \wedge r_1\Gamma^*))\mathbf{U}call(CopyFileA) \wedge r_1\Gamma^*])]))))]\Big)\Big]\Big)\Big)\Big)$

(b)

Fig. 6 **a** Email worm, **b** specification of Email worm

this malicious behavior. This can be specified in SCTPL as follows:

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

This SCTPL formula expresses that the program has a loop such that there are two variables r_1 and r_2, and 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.

3.3.2 Email worms

The typical behavior of an email worm can be summarized as follows: the worm will first call the API *GetModuleFileNameA* 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 6a shows a disassembled fragment of a code corresponding to this typical behavior. This behavior can be expressed by the SCTPL formula of Fig. 6b. 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 $\neg\exists v(mov(r_0, v) \vee lea(r_0, v))\ \mathbf{U}push(r_0)$). 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.

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

Fig. 7 **a** Normal call and **b** obfusated call

3.3.3 Obfuscated calls

Virus writers try to obfuscate their code by, for e.g., hiding the calls to the operating system. For example, a *call* instruction can be replaced by pushes and jumps. Figure 7 shows two equivalent fragments achieving a "call" instruction. Figure 7a 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 Fig. 7b, 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:

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

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.

3.3.4 Obfuscated returns

Virus writers usually obfuscate the returns of their calls to make it difficult to manually or automatically analyze 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. For example, the program in Fig. 8 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:

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

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.

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

Fig. 8 Fragment of the virus Klinge

Fig. 9 A fragement of an appending virus

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

3.3.5 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 Fig. 9 (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.

3.3.6 Spywares

The aim of a spyware is to steal information from the host. To do this, it has to scan the disk of the host to find the interesting file that he wants to steal. If a file is found, it will run a payload to steal it, then continue searching the next file. If a directory is found, it will enter this path and continue scanning. Figure 10 shows a fragment of the notorious spyware Flame. It first calls the function *FindFirstFileW* to search for the first object in the given path, then it will check whether the function call succeeds or not. If the function call fails, it will call the function *GetLastError*. Otherwise it will call either the function *FindFirstFileW* again if it finds a directory or the function *FindNextFileW* with the return value of *FindFirstFileW* as first parameter to search for the next object. We can specify this behavior in SCTPL as follows:

```
l1 : call ds:FindFirstFileW
mov [esi+268h], eax
cmp eax, 0FFFFFFFFh
jnz short l2
...
jnz l1
...
call ds:FindNextFileW
...
l2:call ds:GetLastError
```

Fig. 10 A fragment of Flame

$$\Psi_{spy} = \mathbf{EF}\Big(call(FindFirstFileW) \wedge \mathbf{EX}\ \ \exists x\big(\mathbf{A}[\neg\exists y(mov(eax,y) \vee call(y))\mathbf{U}(mov(x,eax) \wedge \mathbf{AF}\big(call(GetLastError) \vee call(FindFirstFileW) \vee (call(FindNextFileW) \wedge x\Gamma^*)\big))]\big)\Big)$$

This formula states that there exists a control point where the function *FindFirstFileW* is called. Later the value of the register *eax* is not changed until it is assigned to a variable x (note that after a function call, the return value will be stored in the register *eax*). Then, in all the future paths, it either calls *GetLastError* meaning that the function call *FindFirstFileW* fails or calls *FindFirstFileW* when a directory is found or calls *FindNextFileW* when x (the return value of *FindFirstFileW*) is on the top of the stack. A binary code having this behavior may be a benign program, but we can combine this behavior with other malicious behaviors expressing the payload such as sending a file to determine whether the binary code is a malware or not. Note that this formula is branching-time and cannot be expressed in a linear-time temporal logic.

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 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 $\mathcal{B}$ the set of all the environment functions $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$.

4.1 Variable automata

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ s.t. $\Gamma \subseteq \mathcal{D}$, a *Variable Automaton* (VA) is a tuple $M = (Q, \Gamma, \delta, q_0, A)$, where Q is a finite set of states; Γ 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: $p \xrightarrow{\alpha} \{q_1, \ldots, q_n\}$ where α can be x, $\neg x$, or γ, for any $x \in \mathcal{X}$ and $\gamma \in \Gamma$, and $p, q_1, \ldots, q_n \in Q$.

Let $B \in \mathcal{B}$. A run of VA on a word $\gamma_1, \ldots, \gamma_m$ under B is a tree of height m whose root is labeled by the initial state q_0, and each node at depth k labeled by a state q has h children labeled by $p_1, \ldots, p_h$, respectively, such that

– either $q \xrightarrow{\gamma_k} \{p_1, \ldots, p_h\} \in \delta$ and $\gamma_k \in \Gamma$;
– or $q \xrightarrow{x} \{p_1, \ldots, p_h\} \in \delta$, $x \in \mathcal{X}$ and $B(x) = \gamma_k$;
– or $q \xrightarrow{\neg x} \{p_1, \ldots, p_h\} \in \delta$, $x \in \mathcal{X}$ and $B(x) \neq \gamma_k$.

A branch of the tree is accepting iff the leaf of the branch has an accepting state. A run is accepting iff all its branches are accepting. A word $\omega \in \Gamma^*$ 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 $(P \times \Gamma^*) \times \mathcal{B}$. $(\langle p, \omega\rangle, B) \in L(M)$ iff M accepts the word ω under the environment B.

We can show that:

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

The proof is given in Appendix A1. The fact that the transitions of a VA are alternating is crucial to have this closure property. One cannot compute the complement of a VA without using alternating transition rules.

Moreover, we show that every regular variable expression can be effectively represented by a VA:

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

The proof is given in Appendix A2. The construction is similar to the construction of a finite automaton from a regular expression.

4.2 Symbolic Alternating Büchi Pushdown Systems

4.2.1 Definition

A *Symbolic Alternating Büchi Pushdown System* (SABPDS) is a tuple $\mathcal{BP} = (P, \Gamma, \Delta, F)$, where P is a finite set of control locations; $\Gamma \subseteq \mathcal{D}$ is the stack alphabet; $F \subseteq P \times 2^{\mathcal{B}}$ is a set of accepting states; Δ is a finite set of transitions of the form

$$\langle p, \gamma\rangle \overset{\Re}{\hookrightarrow} [\langle p_1, \omega_1\rangle, \ldots, \langle p_n, \omega_n\rangle],$$

where $p \in P$, $\gamma \in \Gamma$, for every i, $1 \le i \le n : p_i \in P$, $\omega_i \in \Gamma^*$, and $\Re : (\mathcal{B})^n \longrightarrow 2^{\mathcal{B}}$ 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 $\omega \in \Gamma^*$ 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 $t = \langle p, \gamma\rangle \overset{\Re}{\hookrightarrow} [\langle p_1, \omega_1\rangle, \ldots, \langle p_n, \omega_n\rangle] \in \Delta$ be a transition, n is the width of the transition t. For every $\omega \in \Gamma^*$, $B, B_1, \ldots, B_n \in \mathcal{B}$, if $B \in \Re(B_1, \ldots, B_n)$, then the configuration $\langle [p, B], \gamma\omega\rangle$ (resp. $\{\langle [p_1, B_1], \omega_1\omega\rangle, \ldots, \langle [p_n, B_n],$

$\omega_n\omega\rangle\}$) is an immediate predecessor (resp. immediate successor) of $\{\langle[p_1, B_1], \omega_1\omega\rangle, \cdots, \langle[p_n, B_n], \omega_n\omega\rangle\}$ (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 $\langle[p_1 B_1], \omega_1\rangle, \ldots, \langle[p_n, B_n], \omega_n\rangle$, respectively, then, necessarily, $\langle[p, B], \omega\rangle$ is an immediate predecessor of $\{\langle[p_1, B_1], \omega_1\rangle, \ldots, \langle[p_n, B_n], \omega_n\rangle\}$ in $\mathcal{BP}$.

A path $c_0c_1\ldots$ 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 $Pre_{\mathcal{BP}} : 2^{(P\times\mathcal{B})\times\Gamma^*} \longrightarrow 2^{(P\times\mathcal{B})\times\Gamma^*}$, $Pre^*_{\mathcal{BP}} : 2^{(P\times\mathcal{B})\times\Gamma^*} \longrightarrow 2^{(P\times\mathcal{B})\times\Gamma^*}$ 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 $Pre^+_{\mathcal{BP}}$.

4.2.2 SABPDS versus ABPDS

An *Alternating Büchi Pushdown System* (ABPDS for short) [28] can be seen as an SABPDS such that $\mathcal{X} = \emptyset$, $\mathcal{D} = \{\perp\}$, and every function $\Re : (\mathcal{B})^n \longrightarrow 2^{\mathcal{B}}$ is of the form $\Re(B_1, \ldots, B_n) = B_\perp$, where $B_\perp(\perp) = \perp$. Such a function will be denoted by $\Re_\perp$. SABPDSs can be simulated by ABPDSs. Indeed, each SABPDS rule of the form

$$\langle p, \gamma\rangle \overset{\Re}{\hookrightarrow} [\langle p_1, \omega_1\rangle, \ldots, \langle p_n, \omega_n\rangle] \in \Delta$$

can be translated into a set of ABPDS rules of the form

$$\langle [\![p, B]\!], \gamma\rangle \overset{\Re_\perp}{\hookrightarrow} [\langle [\![p_1, B_1]\!], \omega_1\rangle, \ldots, \langle [\![p_n, B_n]\!], \omega_n\rangle]$$

where $B, B_1, \ldots, B_n$ can be any elements in $\mathcal{B}$ s.t. $B \in \Re(B_1, \ldots, B_n)$. 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 $|\mathcal{B}| = |D|^{|\mathcal{X}|}$.*

4.2.3 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 and (2) if $q \xrightarrow{\gamma} \{q_1, \ldots, q_n\} \in \delta$ 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 4 The time τ is 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 $\tau = O(\lceil log|\mathcal{D}|\rceil \cdot |\mathcal{X}|)$ [9].

4.2.4 Examples of functions $\Re$

We give some examples of functions $\Re$ below that will be used later.

– $id(B) = \{B\}$, for every $B \in \mathcal{B}$. This is the identity function.
– $equal(B_1, \ldots, B_n)$
$$= \begin{cases} \{B_1\} & \text{if } B_i = B_j \quad \text{for every} \quad 1 \leq i, j \leq n, \\ \emptyset & \text{otherwise.} \end{cases}$$
This function checks whether $B_1, \ldots, B_n$ are equal and returns $\{B_1\}$ (which is equal to $\{B_i\}$ for any i) if this is the case and the emptyset otherwise.
– $meet^x_{\{c_1,\ldots,c_n\}}(B_1, \ldots, B_n)$
$$= \begin{cases} Abs_x(B_1) & \text{if } B_i(x) = c_i \text{ and} \\ & B_i(y) = B_j(y) \quad \text{for} \quad y \neq x, \\ & \text{for every } 1 \leq i, j \leq n, \\ \emptyset & \text{otherwise.} \end{cases}$$
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.

- $join_c^x(B_1, \dots, B_n)$
$$= \begin{cases} \{B_1\} \text{ if } B_i = B_j \quad \text{and} \quad B_i(x) = c, \\ \qquad \text{for every } 1 \le i, \quad j \le n, \\ \emptyset \quad \text{otherwise.} \end{cases}$$
This function checks whether $B_i(x) = c$ for every i. If this is the case, it returns $equal(B_1, \dots, B_n)$, otherwise, it returns the emptyset.
- $join_c^{\neg x}(B_1, \dots, B_n)$
$$= \begin{cases} \{B_1\} \text{ if } B_i = B_j \text{ and } B_i(x) \ne c, \\ \qquad \text{for every } 1 \le i, j \le n, \\ \emptyset \quad \text{otherwise.} \end{cases}$$
This function checks whether $B_i(x) \ne c$ for every i. If this is the case, it returns $equal(B_1, \dots, B_n)$, 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 was to determine whether $\langle p, \omega \rangle \models_\lambda \varphi$, i.e., whether there exists an environment $B \in \mathcal{B}$ s.t. $\langle p, \omega \rangle \models_\lambda^B \varphi$. We proceed as follows: we compute a Symbolic Alternating Büchi Pushdown System $\mathcal{BP}$ s.t. $\langle p, \omega \rangle \models_\lambda^B \varphi$ 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 $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Let $Reg^+(\varphi) = \{e_1, \dots, e_k\}$ and $Reg^-(\varphi) = \{e_{k+1}, \dots, e_m\}$ be the two sets of regular variable expressions[1] that occur in φ. As shown in Theorems 2 and 1, for every i, $1 \le i \le k$ 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, $k < j \le m$ 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 $L(M_{\neg e_j}) = (P \times \Gamma^*) \times \mathcal{B} \backslash L(e_j)$. 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

- $F_1 = \{[(\!|p, a(x_1, \dots, x_n)|\!), \beta] \mid a(x_1, \dots, x_n) \in AP^+(\varphi)$ and $\beta = \{B \in \mathcal{B} \mid p \in \lambda\Big(a\big(B(x_1), \dots, B(x_n)\big)\Big)\}\}$;
- $F_2 = \{[(\!|p, \neg a(x_1, \dots, x_n)|\!), \beta] \mid \neg a(x_1, \dots, x_n) \in AP^-(\varphi)$ and $\beta = \{B \in \mathcal{B} \mid p \notin \lambda\Big(a\big(B(x_1), \dots, B(x_n)\big)\Big)\}\}$;
- $F_3 = P \times cl_{\mathbf{R}}(\varphi) \times \{\mathcal{B}\}$; 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. if $\psi = a(x_1, \dots, x_n)$; $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle (\!|p, \psi|\!), \gamma \rangle \in \Delta'$;
2. if $\psi = \neg a(x_1, \dots, x_n)$; $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle (\!|p, \psi|\!), \gamma \rangle \in \Delta'$;
3. if $\psi = \psi_1 \wedge \psi_2$; $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{equal} [\langle (\!|p, \psi_1|\!), \gamma \rangle, \langle (\!|p, \psi_2|\!), \gamma \rangle] \in \Delta'$;
4. if $\psi = \psi_1 \vee \psi_2$; $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle (\!|p, \psi_1|\!), \gamma \rangle \in \Delta'$ and $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle (\!|p, \psi_2|\!), \gamma \rangle \in \Delta'$;
5. if $\psi = \exists x\ \psi_1$; $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{meet^x_{\{c\}}} \langle (\!|p, \psi_1|\!), \gamma \rangle \in \Delta'$, for every $c \in \mathcal{D}$;
6. if $\psi = \forall x\ \psi_1$; $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{meet^x_{\mathcal{D}}} \underbrace{[\langle (\!|p, \psi_1|\!), \gamma \rangle, \dots, \langle (\!|p, \psi_1|\!), \gamma \rangle]}_{m} \in \Delta'$, where m is the number of elements in $\mathcal{D}$ and $\langle (\!|p, \psi_1|\!), \gamma \rangle$ is repeated m times;
7. if $\psi = \mathbf{EX}\psi_1$; for every $\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta$, $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle (\!|p', \psi_1|\!), \omega \rangle \in \Delta'$;
8. if $\psi = \mathbf{AX}\psi_1$; $\langle (\!|p, \psi)|\!), \gamma \rangle \xhookrightarrow{equal} [\langle (\!|p_1, \psi_1|\!), \omega_1 \rangle, \dots, (\!|p_\ell, \psi_1|\!), \omega_\ell \rangle] \in \Delta'$ such that for every i, $1 \le i \le \ell$, $\langle p, \gamma \rangle \hookrightarrow \langle p_i, \omega_i \rangle \in \Delta$ and these transitions are all the transitions of Δ that have $\langle p, \gamma \rangle$ as left-hand side;
9. if $\psi = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$; $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle (\!|p, \psi_2|\!), \gamma \rangle \in \Delta'$ and $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{equal} [\langle (\!|p, \psi_1|\!), \gamma \rangle, \langle (\!|p', \psi|\!), \omega \rangle] \in \Delta'$ for every transition rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta$;
10. if $\psi = \mathbf{A}[\psi_1 \mathbf{U} \psi_2]$; $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle (\!|p, \psi_2|\!), \gamma \rangle \in \Delta'$, and $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{equal} [\langle (\!|p, \psi_1|\!), \gamma \rangle, \langle (\!|p_1, \psi|\!), \omega_1 \rangle, \dots, \langle (\!|p_\ell, \psi|\!), \omega_\ell \rangle] \in \Delta'$ such that for every i, $1 \le i \le \ell$, $\langle p, \gamma \rangle \hookrightarrow \langle p_i, \omega_i \rangle \in \Delta$ and these transitions are all the transitions of Δ that have $\langle p, \gamma \rangle$ as left-hand side;
11. if $\psi = \mathbf{E}[\psi_1 \mathbf{R} \psi_2]$; $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{equal} [\langle (\!|p, \psi_2|\!), \gamma \rangle, \langle (\!|p, \psi_1|\!), \gamma \rangle] \in \Delta'$, and $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{equal} [\langle (\!|p, \psi_2|\!), \gamma \rangle, \langle (\!|p', \psi|\!), \omega \rangle] \in \Delta'$ for every $\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta$;
12. if $\psi = \mathbf{A}[\psi_1 \mathbf{R} \psi_2]$; $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{equal} [\langle (\!|p, \psi_1|\!), \gamma \rangle, \langle (\!|p, \psi_2|\!), \gamma \rangle] \in \Delta'$, and $\langle (\!|p, \psi|\!), \gamma \rangle \xhookrightarrow{equal} [\langle (\!|p, \psi_2|\!), \gamma \rangle, \langle (\!|p_1, \psi|\!), \omega_1 \rangle, \dots, \langle (\!|p_\ell, \psi|\!), \omega_\ell \rangle] \in \Delta'$ such that for every i, $1 \le i \le \ell$, $\langle p, \gamma \rangle \hookrightarrow \langle p_i, \omega_i \rangle \in \Delta$ and these transitions are all the transitions of Δ that have $\langle p, \gamma \rangle$ as left-hand side;
13. if $\psi = e$: $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle s_e, \gamma \rangle \in \Delta'$, where s_e is the initial state of M_e;
14. if $\psi = \neg e$: $\langle (\!|p, \psi|\!), \gamma \rangle \overset{id}{\hookrightarrow} \langle s_{\neg e}, \gamma \rangle \in \Delta'$, where $s_{\neg e}$ is the initial state of $M_{\neg e}$;
15. for every transition $q \xrightarrow{\alpha} \{q_1, \dots, q_n\}$ in $\mathcal{M}$; $\langle q, \gamma \rangle \overset{\Re}{\hookrightarrow} \{\langle q_1, \epsilon \rangle, \dots, \langle q_n, \epsilon \rangle\} \in \Delta'$, where
    (a) $\Re = equal$ if $\alpha = \gamma$,

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

(b) $\Re = join^x_\gamma$ if $\alpha = x \in \mathcal{X}$,

(c) $\Re = join^{\neg x}_\gamma$ if $\alpha = \neg x$ and $x \in \mathcal{X}$; and

16. for every $q \in \mathcal{F}$; $\langle q, \sharp\rangle \overset{id}{\hookrightarrow} \langle q, \sharp\rangle \in \Delta'$.

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 as follows.

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

If $\psi = \neg a(x_1, \dots, x_n) \in AP^-(\varphi)$, then for every $\omega \in \Gamma^*$, $\langle p, \omega\rangle$ satisfies ψ under any environment B such that $p \notin \lambda(a(B(x_1), \dots, B(x_n)))$. Thus, for such B, $\mathcal{BP}_\varphi$ should have an accepting run from the configuration $\langle[(\!|p, \neg a(x_1, \dots, x_n)|\!), B], \omega\rangle$. This is ensured by Item 2 that adds a loop in $\langle[(\!|p, \neg a(x_1, \dots, x_n)|\!), B], \omega\rangle$ (all accepting paths are infinite). The definition of F_2 guarantees that the state $[(\!|p, \neg a(x_1, \dots, x_n)|\!), B]$ is accepting.

If $\psi = \psi_1 \wedge \psi_2$, Item 3 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 4 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 5 since $B \in meet^x_{\{c\}}\big(B[x \leftarrow c]\big)$.

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 6 and its corresponding function $meet^x_\mathcal{D}$; if $\mathcal{D} = \{c_1, \dots, c_m\}$, then $B \in meet^x_\mathcal{D}\big(B[x \leftarrow c_1], \dots, B[x \leftarrow c_m]\big)$.

If $\psi = \mathbf{EX}\psi_1$, 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 7 where the function id guarantees that the environment remains the same.

If $\psi = \mathbf{AX}\psi_1$, 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 8 expresses this. The function *equal* makes sure that all these environments are the same.

If $\psi = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$, 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 9. The case $\psi = \mathbf{A}[\psi_1 \mathbf{U} \psi_2]$ is analogous.

If $\psi = \mathbf{E}[\psi_1 \mathbf{R} \psi_2]$, 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 11. 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 $\psi = \mathbf{A}[\psi_1 \mathbf{R} \psi_2]$ 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 13, 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 Item 15. Item 15 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 15(a) deals with the case where $q \xrightarrow{\gamma} \{q_1, \dots, q_2\}$ is a transition in δ_e. In this case, $\mathcal{BP}_\varphi$ moves to the next states $[q_1, B], \dots, [q_n, B]$ 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 15(b) deals with the case where $q \xrightarrow{x} \{q_1, \dots, q_2\}$, $x \in \mathcal{X}$ 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 $[q_1, B], \dots, [q_n, B]$ 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 15(c) deals with the case where $q \xrightarrow{\neg x} \{q_1, \dots, q_2\}$ 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 $[q_1, B], \dots, [q_n, B]$ and pops γ from the stack. The func-

tion $join_{\gamma}^{\neg x}$ 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 $f_1, \ldots, f_n$ of M_e after reading the word w, i.e., iff $\mathcal{BP}_\varphi$ reaches a set of states $[f_1, B], \ldots, [f_n, B]$ 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 16 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.

Remark 5 Esparza et al. introduced two approaches to reduce LTL with regular valuations model checking for PDSs to LTL model checking for PDSs [14,15]. Our approach is more direct, since we do not need to compute the product of the PDS with the finite automata representing the predicates over the stack.

Formally, we can show the following theorem (Proof refers to Appendix A3):

Theorem 3 *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a function $\lambda : AP_{\mathcal{D}} \longrightarrow 2^P$, a SCTPL formula φ, and a configuration $\langle p, \omega\rangle$ of $\mathcal{P}$, we have for every $B \in \mathcal{B}$, $\langle p, \omega\rangle \models_\lambda^B \varphi$ 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.

4.4.1 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 $\mathcal{BP} = (P, \Gamma, \Delta, F)$, $\mathcal{L}(\mathcal{BP}) = Y_{\mathcal{BP}}$.*

Proof (sketch) The proof follows the lines of [28], 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 $P \times \mathcal{B} \times \Gamma^*$. □

4.4.2 Computing $Y_{\mathcal{BP}}$

Our goal was 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 [28] that computes an AMA recognizing the language of an ABPDS. We show that $Y_{\mathcal{BP}}$ can be represented by a SAMA $\mathcal{A} = (\mathcal{Q}, \Gamma, \delta, I, \mathcal{Q}_f)$ whose set of states $\mathcal{Q}$ 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 ($\mathcal{Q}_f = \{q_f\}$). 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 $\Re : (\mathcal{B})^n \longrightarrow 2^{\mathcal{B}}$ to $\Re : (2^{\mathcal{B}})^n \longrightarrow 2^{\mathcal{B}}$ as follows: $\Re(\beta_1, \ldots, \beta_n) = \{B \in \mathcal{B} \mid B \in \Re\{B_1, \ldots, B_n\}$ s.t. for every $1 \leq i \leq n : B_i \in \beta_i\}$; and we define two functions π^{-1} and π^i as follows: For every $S \subseteq \mathcal{Q}$,

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

Algorithm 1 shown in Table 1 computes a SAMA $\mathcal{A}$ recognizing $Y_{\mathcal{BP}}$. To understand the idea behind this algorithm, let $\mathbf{A}_0$ be the automaton obtained after the initialization step and $\mathbf{A}_i$ 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., $\mathbf{A}_i$ 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 $\mathbf{A}_0$ 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 $[p, \beta' \cap \beta]^i \xrightarrow{\epsilon} [p, \beta']^{i-1}$, $s.t.$ $[p, \beta']^{i-1} \xrightarrow{\gamma} Q \in \delta$ for every $[p, \beta] \in F$. After this step, we obtain $L(\mathbf{A}_{i-1}) \cap F \times \Gamma^*$. $loop_2$ (Lines 4–8) is the saturation procedure that computes the Pre^* of $L(\mathbf{A}_{i-1}) \cap F \times \Gamma^*$. Line 9 removes the ϵ-transition added by Line 3. After Line 9, the automaton $\mathbf{A}_i$ recognizes $Pre^+(L(\mathbf{A}_{i-1}) \cap F \times \Gamma^*)$. 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:

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

Proof (sketch) The proof follows the lines of the proof of [28]. Indeed, our algorithm follows the idea of the algorithm that computes an AMA recognizing the language of an ABPDS given in [28]. 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 $\mathcal{X}$.

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

```
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 $\langle p,\gamma \rangle \overset{\Re}{\hookrightarrow} [\langle p_1,\omega_1 \rangle, ..., \langle p_n,\omega_n \rangle]$ in $\Delta$,
6.           and every case where $[p_k,\beta_k]^i \xrightarrow{\omega_k}_\delta Q_k$, for all $1 \le k \le n$;
7.               Add a new rule $[p,\beta]^i \xrightarrow{\gamma} \bigcup_{k=1}^n Q_k$ in $\delta$ where $\beta = \Re(\beta_1, ..., \beta_n)$;
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 $[p,\beta]^{i-1} \xrightarrow{\gamma} \pi^{-1}(R) \in \delta$
```

2. To compute the Pre^* of $L(\mathbf{A}_{i-1}) \cap F \times \Gamma^*$, instead of using the following saturation procedure given in [7] that computes the Pre^* for alternating pushdown systems:

   If $\langle p, \gamma \rangle \hookrightarrow \{\langle p_1, \omega_1 \rangle, \ldots, \langle p_n, \omega_n \rangle\}$ and $p_k^i \xrightarrow{\omega_k}_\delta Q_k$ for all $1 \le k \le n$,
   add a transition $p^i \xrightarrow{\gamma} \bigcup_{k=1}^n Q_k$.

   We use the following saturation procedure:

   If $\langle p, \gamma \rangle \overset{\Re}{\hookrightarrow} [\langle p_1, \omega_1 \rangle, \ldots, \langle p_n, \omega_n \rangle]$ and $[p_k, \beta_k]^i \xrightarrow{\omega_k}_\delta Q_k$ for all $1 \le k \le n$, add a transition $[p, \beta]^i \xrightarrow{\gamma} \bigcup_{k=1}^n Q_k$, where $\beta = \Re(\beta_1, \ldots, \beta_n)$.

   Indeed, intuitively, if $\langle [p, B], \gamma\omega \rangle$ 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 $\langle p, \gamma \rangle \overset{\Re}{\hookrightarrow} [\langle p_1, \omega_1 \rangle, \langle p_2, \omega_2 \rangle]$, and $\langle [p_1, B_1], \omega_1\omega \rangle$ and $\langle [p_2, B_2], \omega_2\omega \rangle$ are in $L(\mathbf{A}_{i-1}) \cap F \times \Gamma^*$, then, necessarily, $B \in \Re(B_1, B_2)$ and there exist $\beta_1, \beta_2 \subseteq \mathcal{B}$ and $S_1, S_2 \subseteq \mathcal{Q}$ 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 $[p, \Re(\beta_1, \beta_2)] \xrightarrow{\gamma}_\delta S_1 \cup S_2$. This allows accepting the configuration $\langle [p, B], \gamma\omega \rangle$ using the run $[p, \Re(\beta_1, \beta_2)] \xrightarrow{\gamma}_\delta S_1 \cup S_2 \xrightarrow{\omega}_\delta q_f$.
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 $L(\mathbf{A}_{i-1}) \cap F \times \Gamma^*$ 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 $\mathbf{A}_{i-1}$ 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 $[p, \beta'] \notin F$. □

Complexity: 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, [30] provides a way to efficiently implement the saturation procedure of [7] that computes the Pre^* of $L(A)$ 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 $\Re$ 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 [30]. Thus, by adapting the complexity analysis of [30], 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 $O\left(|P|^2 \cdot 2^{2|\mathcal{B}|} \cdot |\Gamma| \cdot |\Delta| \cdot 2^{5|P| \cdot 2^{|\mathcal{B}|}}\right)$.

Thus, we get:

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 $O(|P|^2 \cdot 2^{2|\mathcal{B}|} \cdot |\Gamma| \cdot |\Delta| \cdot 2^{5|P| \cdot 2^{|\mathcal{B}|}})$ time.*

Remark 6 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 [28] 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 can be compactly represented using BDDs for example, whereas in the explicit case (for ABPDS), all the environments have to be considered. Thus, **Algorithm 1** will behave better in practice. This is illustrated in Sect. 5. Indeed, in the experiments we run, in the majority of cases, **Algorithm 1** terminates in few seconds, whereas if we compute an equivalent ABPDS and apply the algorithm of [28], we run out of memory.

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 $O((|P||\varphi| + k)^2 \cdot 2^{2|\mathcal{B}|} \cdot |\Gamma| \cdot ((|P||\Gamma| + |\Delta|)|\varphi| + d) \cdot 2^{5(|P||\varphi|+k)\cdot 2^{|\mathcal{B}|}})$, 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 *One can decide whether a PDS configuration satisfies a SCTPL formula in time $O\Big((|P||\varphi| + k)^2 \cdot 2^{2|\mathcal{B}|} \cdot |\Gamma| \cdot \big((|P||\Gamma| + |\Delta|)|\varphi| + d\big) \cdot 2^{5(|P||\varphi|+k)\cdot 2^{|\mathcal{B}|}}\Big)$.*

Remark 7 As described in Remark 3, 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 [28], 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 using BDDs, model-checking SCTPL using our symbolic techniques behaves much better than reducing SCTPL to CTL with regular valuations and then applying [28]. Indeed, the experiments in Sect. 5 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 [28] would run out of memory.

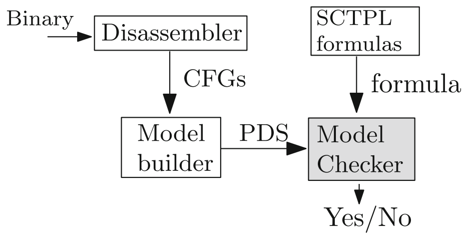

Fig. 11 Overview of our framework

5 Experiments

We implemented our techniques in a tool for malware detection. Our framework consists of four components: a disassembler, a model builder, a set of SCTPL formulas, and a model checker as shown in Fig. 11. The disassembler extracts a set of control flow graphs (CFGs) from the binary code. In our experiments, we use IDA Pro [19] as disassembler. The model builder translates the CFGs to a Pushdown system (PDS) as described in Sect. 2. The set of SCTPL formulas consists of a set of SCTPL specifications of malicious behaviors. The model checker determines whether the PDS satisfies one of the SCTPL specifications. It outputs **Yes** if the PDS satisfies one of the specifications, i.e., if the binary code contains some malicious behaviors, and **No** otherwise. We carried out different experiments. We obtained interesting results. In particular, our tool was able to detect several viruses taken from [18]. Note that in its current form, our tool can detect only viruses that satisfy the specifications described in this paper. Several other viruses could not be detected by our tool, since they have other specifications. To be able to detect them, we need to add their malicious specifications into our tool.

5.1 Symbolic versus 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 6 and 7, this can be done differently in two ways: (1) either translate the SABPDS into an equivalent ABPDS and then apply the algorithm of [28] 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

Table 2 Our techniques versus explicit techniques

PDS	SCTPL size	$\|\mathcal{X}\|$	$\|\mathcal{D}\|$	Our techniques				SABPDS→ABPDS				SCTPL→CTLr	
$\|P\|+\|\Gamma\|+\|\Delta\|$				SABPDS $\|\Delta_s\|$	SAMA $\|\delta_s\|$	Time (s)	Mem (Kb)	ABPDS $\|\Delta_1\|$	AMA $\|\delta_1\|$	Time (s)	Mem (kb)	Time (s)	Mem (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**

Bold highlights the improvements of my symbolic techniques

an existing algorithm for model-checking PDSs against CTL with regular valuations (such as the one given in [28]). In order to show that our approach is much better than these two solutions, we run several experiments that compare the three approaches. The results are summarized in Table 2. **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. **Columns** under "Our techniques" describe the results obtained using our techniques. **Columns** under "SABPDS→ABPDS" describe the results obtained when we translated the SABPDS to an equivalent ABPDS and then applied the algorithm of [28]. **Columns** under "SCTPL→CTLr" describe the results obtained when the SCTPL formula is translated into a CTL with regular valuations formula, and then used the algorithm of [28]. $|\Delta_s|$ 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 [28]. **Columns** *time(s)* and *mem(kb)* give the time (in seconds) and the memory (in kilobytes). **Memout** means "memory out" (the memory limit is 650 Mb), respectively. The results described in Table 2 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. Using CTLr model-checking techniques costs a lot of time. This is due to the fact that the obtained CTLr formulas are large as described in Remark 3.

5.2 Malware detection

We applied our tool to detect several malicious programs. Our results are reported in Table 3. **Column** *LOC* gives the number of lines of code in x86 assembly programs. **Column** *Formula* describes the malicious behavior specification that allowed to detect that the program is a malware. We considered the SCTPL specifications described in Sect. 3.3: ψ_{wv} for Kernel32.dll base address viruses, ψ_{ew} for email worms, ψ_{or} for viruses using obfuscated returns (like appending viruses), and ψ_{oc} for viruses using obfuscated calls. ψ'_{ew} is a CTPL formula that can specify email worms in a less precise manner (without taking into account the stack) as described in [20–22]. Every program is checked against all these specifications. A program is declared as a potential virus if it satisfies one of these specifications (some viruses like Alcaul.i and Alcaul.j can be detected by several specifications). **Columns** *time(s)* and *mem(Mb)* give the time (in seconds) and the memory (in Mb), respectively. The last **Column** *result* is *Y* if the program contains the malicious behaviors given in **Column** *Formula*, and *N* if not. We also compared our techniques against translating SABPDS to ABPDS, or translating SCTPL to CTL with regular valuations. We were able to detect all the viruses that we considered, whereas applying the translation from SABPDS to ABPDS or from SCTPL to

Table 3 Detection of real malwares

Examples	LOC	Formula	Our techniques		SABPDS→ABPDS		SCTPL→CTLr		Result
			Time(s)	Mem(Mb)	Time(s)	Mem(Mb)	Time(s)	Mem(Mb)	
Windows virus									
Adson.1559	32	Ψ_{wv}	0.22	2.1	–	MemOut	–	MemOut	Y
Adson.1651	33	Ψ_{wv}	0.23	2.1	–	MemOut	–	MemOut	Y
Adson.1703	32	Ψ_{wv}	0.25	2.1	–	MemOut	–	MemOut	Y
Adson.1734	40	Ψ_{wv}	0.31	2.6	–	MemOut	–	MemOut	Y
Alcaul.d	32	Ψ_{wv}	0.20	0.8	–	MemOut	47.70	51	Y
Alcaul.i	40	Ψ_{wv}	4.38	0.28	–	MemOut	159.88	169.64	Y
Alcaul.j	48	Ψ_{wv}	0.30	2.1	–	MemOut	218.25	198.71	Y
Email Worm									
Klez.a	32	Ψ'_{ew}	1.62	10.8	–	MemOut	–	MemOut	Y
Klez.b	25	Ψ'_{ew}	1.55	10.8	–	MemOut	–	MemOut	Y
Klez.c	25	Ψ'_{ew}	1.27	8.9	–	MemOut	–	MemOut	Y
Klez.d	26	Ψ'_{ew}	1.47	10.3	–	MemOut	–	MemOut	Y
Klez.e	22	Ψ'_{ew}	0.77	7.0	–	MemOut	–	MemOut	Y
Klez.f	22	Ψ'_{ew}	0.76	7.0	–	MemOut	–	MemOut	Y
Klez.g	22	Ψ'_{ew}	0.75	7.0	–	MemOut	–	MemOut	Y
Klez.i	22	Ψ'_{ew}	0.74	7.0	–	MemOut	–	MemOut	Y
Klez.j	22	Ψ'_{ew}	0.74	7.0	–	MemOut	–	MemOut	Y
Mydoom.c	153	Ψ'_{ew}	145.20	322.8	–	MemOut	–	MemOut	Y
Mydoom.e	130	Ψ'_{ew}	123.22	267.5	–	MemOut	–	MemOut	Y
Mydoom.g	127	Ψ'_{ew}	117.50	256.7	–	MemOut	–	MemOut	Y
Netsky.a	40	Ψ'_{ew}	573.8	10.1	–	MemOut	–	MemOut	Y
Netsky.a	40	Ψ_{ew}	2.73	14.5	–	MemOut	–	MemOut	Y
Netsky.b	40	Ψ'_{ew}	573.8	10.1	–	MemOut	–	MemOut	Y
Netsky.b	40	Ψ_{ew}	2.73	14.5	–	MemOut	–	MemOut	Y
Netsky.c	40	Ψ'_{ew}	573.8	10.1	–	MemOut	–	MemOut	Y
Netsky.c	40	Ψ'_{ew}	2.73	14.5	–	MemOut	–	MemOut	Y
Netsky.d	40	Ψ'_{ew}	573.8	10.1	–	MemOut	–	MemOut	Y
Netsky.d	40	Ψ_{ew}	2.73	14.5	–	MemOut	–	MemOut	Y

Table 3 continued

Examples	LOC	Formula	Our techniques		SABPDS→ABPDS		SCTPL→CTLr		Result
			Time(s)	Mem(Mb)	Time(s)	Mem(Mb)	Time(s)	Mem(Mb)	
Obfuscated return									
Akez	16	Ψ_{or}	0.22	0.3	–	MemOut	0.44	2.49	Y
Alcaul.b	10	Ψ_{or}	0.06	0.2	–	MemOut	0.28	1.18	Y
Alcaul.c	13	Ψ_{or}	0.12	0.3	–	MemOut	0.41	2.19	Y
Alcaul.e	19	Ψ_{or}	0.49	0.9	–	MemOut	1.03	5.28	Y
Alcaul.f	15	Ψ_{or}	0.09	0.3	–	MemOut	0.53	2.23	Y
Alcaul.g	18	Ψ_{or}	0.31	0.7	–	MemOut	0.97	4.45	Y
Alcaul.h	21	Ψ_{or}	0.83	0.9	–	MemOut	1.14	6.88	Y
Alcaul.i	40	Ψ_{or}	54.92	1.17	–	MemOut	155.94	169.65	Y
Alcaul.j	48	Ψ_{or}	1.41	9.6	–	MemOut	190.39	198.71	Y
Alcaul.k	17	Ψ_{or}	0.26	0.6	–	MemOut	0.76	3.65	Y
Alcaul.l	19	Ψ_{or}	0.30	0.7	–	MemOut	0.86	3.96	Y
Alcaul.m	18	Ψ_{or}	0.20	0.6	–	MemOut	0.88	3.37	Y
Alcaul.n	12	Ψ_{or}	0.12	0.3	–	MemOut	0.44	2.28	Y
Alcaul.o	22	Ψ_{or}	0.20	0.6	–	MemOut	0.83	3.37	Y
Klinge	34	Ψ_{or}	237.50	4.49	–	MemOut	0.83	3.37	Y
Evol.a	45	Ψ_{or}	9.58	3.22	–	MemOut	–	MemOut	Y
Obfuscated call									
Oroch.3982	24	Ψ_{oc}	3.70	7.72	–	MemOut	–	MemOut	Y
KME	99	Ψ_{oc}	999.31	20.04	–	MemOut	–	MemOut	Y
Anar.a	12	Ψ_{oc}	1.16	1.60	885.33	343.24	54.92	34.12	Y
Anar.b	12	Ψ_{oc}	1.49	1.60	891.42	348.54	56.14	36.16	Y
Atak.b	87	Ψ_{oc}	762.34	18.15	–	MemOut	–	MemOut	Y
Predec.j	20	Ψ_{oc}	0.23	0.81	–	MemOut	56.14	36.16	Y
Bagle.d	56	Ψ_{oc}	652.23	16.96	–	MemOut	–	MemOut	Y
Benign binary code									
Uedit32	91	Ψ_{wv}	0.53	5.74	–	MemOut	–	MemOut	N
Uedit32	91	Ψ_{or}	5.34	28.44	–	MemOut	–	MemOut	N
Uedit32	91	Ψ_{ew}	21.12	111.84	–	MemOut	–	MemOut	N
Uedit32	91	Ψ_{oc}	92.58	100.94	–	MemOut	–	MemOut	N
Cygwin32	67	Ψ_{wv}	0.30	5.21	–	MemOut	–	MemOut	N
Cygwin32	67	Ψ_{or}	5.70	30.44	–	MemOut	–	MemOut	N
Cygwin32	67	Ψ_{ew}	23.72	123.31	–	MemOut	–	MemOut	N
Cygwin32	67	Ψ_{oc}	45.80	180.42	–	MemOut	–	MemOut	N
cmd.exe	100	Ψ_{wv}	1.44	25.52	–	MemOut	–	MemOut	N
cmd.exe	100	Ψ_{or}	325.78	330.67	–	MemOut	–	MemOut	N
cmd.exe	100	Ψ_{ew}	118.11	335.88	–	MemOut	–	MemOut	N
cmd.exe	100	Ψ_{oc}	1035.52	250.11	–	MemOut	–	MemOut	N

CTL with regular valuations would run out of memory in most of the cases, and thus cannot detect the viruses. It can also be observed that when both the SCTPL formula ψ_{ew} and the CTPL formula ψ'_{ew} are satisfied (e.g., for the variants of the Netsky virus), then the time consumption using SCTPL is less than that using CTPL. Table 3 shows that our approach is able to efficiently detect several 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.

5.3 Obfuscated viruses

We run several experiments to check how robust our techniques are for malware detection in case the virus writers use obfuscation techniques. To this aim, we considered some of the viruses of Table 3, 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 five 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 4. Our techniques were able to detect all these variations, whereas the three well known and widely used free antiviruses *Avira* [2], and *Qihoo 360* [26] and *Avast* [1], were not able to detect several of these virus variations. Moreover, as described in Table 4, some of the viruses we tried to obfuscate could also be detected by a less precise CTPL formula. When we add obfuscation based on stack operations to these viruses, they cannot be detected by this CTPL formula anymore, whereas SCTPL is still able to detect them. (Note that the majority of the malwares that we detect in Table 4 need a SCTPL formula to get detected, whereas CTPL is not sufficient to detect them.)

6 Related work

These last years, there has been a substantial amount of research to find efficient techniques that can detect viruses. A lot of techniques use signature-based or emulation-based approaches. As already mentioned in Sect. 1, such techniques have some limitations. Indeed, signature matching fails if the virus does not use a known signature. As for emulation techniques, they can execute the program only in a given time interval and execute only one trace (while model-checking techniques can check all the possible traces). Thus, emulation-based techniques can miss the malicious behaviors if they occur after the timeout. Moreover, malwares commonly use anti-emulation techniques. For instance, a malware executes its malicious behaviors at some specific day or time. Also, a malware can identify whether it is running in a virtual environment by checking the CPU cycles used for executing some instructions. If it is running in a virtual environment, it can stop its malicious behaviors.

Model-checking and static analysis techniques have been applied to detect malicious behaviors, e.g., in [5,10,12,20–22,27]. 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. Bonfante et al. [6] use tree automata to represent a set of malicious behaviors. However, [6] cannot specify predicates over the stack content.

Lakhotia et al. [25] keep 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 obfuscated calls and obfuscated returns. However, they cannot specify the other malicious behaviors that we are able to detect using our SCTPL specifications.

Lakhotia et al. [24] perform 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 [24] were only tried on toy examples, they have not been applied for malware detection.

Balakrishnan et al. [4] use pushdown systems for binary code analysis. However, the translation from programs to PDSs in [4] assumes that the program follows a standard compilation model where calls and returns match. As we have shown, several malicious behaviors do not follow this model. Our translation from a control flow graph to a PDS does not make this assumption. They can detect situations where calls and returns are not matched. However, they consider only reachability. Our SCTPL logic allows specifying a larger set of malicious behaviors.

Stack computation tree predicate logic can be seen as an extension of CTPL with predicates over the stack content. CTPL was introduced in [20–22]. In these works, the authors show how CTPL can be used to succinctly specify

Table 4 Detection of obfuscated viruses

Obfuscation	Our techniques detection rate (%)	Avira antivirus detection rate (%)	Qihoo 360 antivirus detection rate (%)	Avast antivirus detection rate (%)	CTPL model-checking detection rate (%)
Nop-insertion	100	65	55	60	80
Code-reordering	100	40	35	45	75
Register-renaming	100	25	25	30	50
Stack-operation	100	20	25	20	0
Procedure-split	100	5	5	5	75

malicious behaviors. Our SCTPL logic is more expressive than CTPL. Indeed, CTPL cannot specify predicates over the stack. Thus, SCTPL allows specifying more malicious behaviors than CTPL. Indeed, most of the malicious behaviors we considered cannot be expressed in CTPL. Christodorescu et al. [11] use a kind of graph *malspec* to express malicious behaviors which can specify the dependence of parameters between functions. It can express malicious behavior as a safety property. However, it cannot express malicious behavior as a liveness property (e.g., Ψ_{or}).

7 Conclusion

In this work, we propose a novel approach to detect malware by pushdown systems model checking. First, we propose a new technique to translate binary code to pushdown systems. Our technique is different from other translations from programs to PDSs since it does not need to assume that the program has matched calls and returns. Then, we introduce our SCTPL and show how it can precisely and succinctly specify malicious behaviors that cannot be specified by other existing specification formalisms. We then provide an algorithm to model-check pushdown systems against SCTPL specifications. Our approach consists in reducing this model-checking problem to checking the emptiness of Symbolic Alternating Büchi Pushdown Systems. We implemented our techniques in a tool for malware detection. We obtained encouraging experimental results. As mentioned previously, our approach works if the data in the stack cannot be changed by direct memory access. Since most interesting malicious behaviors consider the occurrences of API function calls and their parameters, our technique still works properly if the direct manipulation of the stack content does not change the value of the parameters that the malicious specifications involved nor change other values that could affect whether the API function calls of the specification are reached or not. If this is not satisfied by the program, our approach will not work. To overcome this limitation, we could extend the PDS model with transitions that modify the whole stack content with a transducer as done in [31], and model binary codes using this new formalism. We plan to investigate this idea in the future.

Appendix

A.1 Proof of Theorem 1

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

Proof We need to prove that variable automata are closed under union, complementation and intersection.

Union. Computing the union of two VAs can be done as for finite automata. Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, let $M^1 = (Q^1, \Gamma, \delta^1, q_0^1, A^1)$ and $M^2 = (Q^2, \Gamma, \delta^2, q_0^2, A^2)$ be two VAs, we can construct a VA M as usual, such that $L(M) = L(M_1) \cup L(M_2)$.

W.l.o.g., we suppose that $Q^1 \cap Q^2 = \emptyset$, otherwise we can rename these repeated states. Let $M = (Q, \Gamma, \delta, q_0, A)$ such that

- $Q = Q^1 \cup Q^2 \cup \{q_0\}$, where q_0 is an additional initial state of M;
- $A = A^1 \cup A^2$;
- $\delta = \delta^1 \cup \delta^2 \cup \{q_0 \xrightarrow{\epsilon} \{q_0^1\}, q_0 \xrightarrow{\epsilon} \{q_0^2\}\}$.

Thus, we obtain that $L(M) = L(M_1) \cup L(M_2)$. ϵ-transitions can be removed as usual.

Complementation. Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, let $M = (Q, \Gamma, \delta, q_0, A)$ be a VA , we construct a VA $\overline{M}$ such that $L(\overline{M}) = (P \times \Gamma^*) \times \mathcal{B} \backslash L(M)$.

W.l.o.g., we assume that if either $q \xrightarrow{x} \{p_1, \ldots, p_h\} \in \delta$ or $q \xrightarrow{\neg x} \{p_1, \ldots, p_h\} \in \delta$, then there does not exist any other transition rule in the form of $q \xrightarrow{\alpha} \{q_1, \ldots, q_n\}$ in δ. Indeed, if there exist two transition rules $q \xrightarrow{x} \{p_1, \ldots, p_n\}$ and $q \xrightarrow{\alpha} \{q_1, \ldots, q_m\}$, or $q \xrightarrow{\neg x} \{p_1, \ldots, p_n\}$ and $q \xrightarrow{\alpha} \{q_1, \ldots, q_m\}$, we then can replace these two transition rules by $q' \xrightarrow{x} \{p_1, \ldots, p_n\}$ and $q'' \xrightarrow{\alpha} \{q_1, \ldots, q_m\}$, or $q' \xrightarrow{\neg x} \{p_1, \ldots, p_n\}$ and $q'' \xrightarrow{\alpha} \{q_1, \ldots, q_m\}$, and replace all the transition rules of the form $g \xrightarrow{\alpha'} \{q, g_1, \ldots, g_h\}$ by two transition rules $g \xrightarrow{\alpha'} \{q', g_1, \ldots, g_h\}$ and $g \xrightarrow{\alpha'} \{q'', g_1, \ldots, g_h\}$.

Let $\overline{M} = (Q \cup \{q_f\}, \Gamma, \delta', q_0, A')$ be a VA, where $q_f \notin Q$ is a final state, $A' = Q \cup \{q_f\} \backslash A$, and δ' is the smallest set of transition rules satisfying the following: for every $p, q \in Q$, $x \in \mathcal{X}$, $\gamma \in \Gamma$,

1. if $p \xrightarrow{x} \{q_1, \ldots, q_m\} \in \delta$; then $p \xrightarrow{x} q_i \in \delta'$ for every $1 \leq i \leq m$, and $p \xrightarrow{\neg x} q_f \in \delta'$,
2. if $p \xrightarrow{\neg x} \{q_1, \ldots, q_m\} \in \delta$; then $p \xrightarrow{\neg x} q_i \in \delta'$ for every $1 \leq i \leq m$, and $p \xrightarrow{x} q_f \in \delta'$,
3. if there does not exist $S \subseteq Q$ s.t. neither $p \xrightarrow{\gamma} S \in \delta$, nor $p \xrightarrow{x} S \in \delta$ nor $p \xrightarrow{\neg x} S \in \delta$; then $p \xrightarrow{\gamma} q_f \in \delta'$,
4. $p \xrightarrow{\gamma} \{q_1, \ldots, q_m \mid$ for every $1 \leq i \leq m : p \xrightarrow{\gamma} S_i \in \delta$ and $q_i \in S_i\} \in \delta'$,
5. $q_f \xrightarrow{\gamma} q_f \in \delta'$.

Let us show that $L(\overline{M}) = (P \times \Gamma^*) \times \mathcal{B} \backslash L(M)$.

($\Longrightarrow$) First we show that $L(\overline{M}) \subseteq (P \times \Gamma^*) \times \mathcal{B} \backslash L(M)$. It is sufficient to prove that for every $(\langle p, \omega \rangle, B) \in (P \times \Gamma^*) \times \mathcal{B}$, if $\overline{M}$ has an accepting run from a state $f \in Q$ on the word ω under the environment B, then M does not have any accepting run from the state f on the word ω under the environment B. We proceed by induction on the length $|\omega|$ of ω.

- **Basis** $|\omega| = 0$: Then $\omega = \epsilon$. Since $\overline{M}$ has an accepting run from the state f on the word ω under the environment B, we obtain that the initial state $f \in A'$. Since $A' = Q \cup \{q_f\}\backslash A$, we get that $f \notin A$ which implies that M does not have any accepting run from the state f on the word ω under the environment B.
- **Step**:$|\omega| \geq 1$: Let $\gamma \in \Gamma, u \in \Gamma^*$ such that $\omega = \gamma u$. $\overline{M}$ has an accepting run from the state f on the word ω under the environment B. Let $t \in \delta'$ be the first transition used by this run, and the left side of t is the state f and the input is γ. The proof depends on the reasoning of the transition rule t added by the above construction.

  - **Case 1**: $t = f \xrightarrow{x} q_i \in \delta'$ is added by Item 1, then we get that $\overline{M}$ has an accepting run from the state q_i on the word u under the environment B and $B(x) = \gamma$. By applying the induction hypothesis, we obtain that M does not have any accepting run from the state q_i on the word u under the environment B.
    Since the transition rule $t = f \xrightarrow{x} q_i \in \delta'$ is added by Item 1, then we get that M has only one transition $f \xrightarrow{x} \{q_i, p_1, \ldots, p_m\} \in \delta$ from the state f due to the assumption, since M does not have any accepting run from the state q_i on the word u under the environment B, we obtain that M does not have any accepting run from the state f on the word γu under the environment B.
  - **Case 2**: $t = f \xrightarrow{\neg x} q_i \in \delta'$ is added by Item 2, then we get that $\overline{M}$ has an accepting run from the state q_i on the word u under the environment B and $B(x) \neq \gamma$. By the induction hypothesis, we obtain that M does not have any accepting run from the state q_i on the word u under the environment B.
    Since the transition rule $t = f \xrightarrow{\neg x} q_i \in \delta'$ is added by Item 2, then we get that M has only one transition $f \xrightarrow{\neg x} \{q_i, p_1, \ldots, p_m\} \in \delta$ from the state f due to the assumption, since M does not have any accepting run from the state q_i on the word u under the environment B, we obtain that M does not have any accepting run from the state f on the word γu under the environment B.
  - **Case 3**: $t = f \xrightarrow{\neg x} q_f \in \delta'$ is added by Item 1, then we get that $B(x) \neq \gamma$ and M has only one transition $f \xrightarrow{x} \{p_1, \ldots, p_m\} \in \delta$ from the state f due to the assumption. Since $B(x) \neq \gamma$, we get that M does not have any accepting run from the state f on the word γu under the environment B.
  - **Case 4**: $t = f \xrightarrow{x} q_f \in \delta'$ is added by Item 2, then we get that $B(x) = \gamma$ and M has only one transition $f \xrightarrow{\neg x} \{p_1, \ldots, p_m\} \in \delta$ from the state f due to the assumption. Since $B(x) = \gamma$, we get that M does not have any accepting run from the state f on the word γu under the environment B.
  - **Case 5**: $t = f \xrightarrow{\gamma} q_f \in \delta'$ is added by Item 3, then M does not have any transition in the form of $f \xrightarrow{\gamma} S \in \delta$, $f \xrightarrow{x} S \in \delta$ or $f \xrightarrow{\neg x} S \in \delta$ for any $S \subseteq Q$, and we get that M does not have any accepting run from the state f on the word γu under the environment B.
  - **Case 6**: $t = f \xrightarrow{\gamma} \{q_1, \ldots, q_m\} \in \delta'$ is added by Item 4, then M has transitions $f \xrightarrow{\gamma} S_i \in \delta$ such that $q_i \in S_i$ for every $1 \leq i \leq m$. Since $\overline{M}$ has an accepting run from the state f on the word ω under the environment B, we obtain that $\overline{M}$ has an accepting run from every state q_i on the word u under the environment B. By applying the induction hypothesis, we obtain that M does not have any accepting run from the state q_i on the word u under the environment B for every $1 \leq i \leq m$. Since M has transitions $f \xrightarrow{\gamma} S_i \in \delta$ for every $1 \leq i \leq m$ and $q_i \in S_i$, then each run from the state f has to go through a state q_i for some $1 \leq i \leq m$, we get that M does not have any accepting run from the state f on the word γu under the environment B.

($\Longleftarrow$) We show that $L(\overline{M}) \supseteq (P \times \Gamma^*) \times \mathcal{B}\backslash L(M)$. It is sufficient to prove that for every $(\langle p, \omega\rangle, B) \in (P \times \Gamma^*) \times \mathcal{B}$, if M does not have any accepting run from a state $f \in Q$ on the word ω under the environment B, then $\overline{M}$ has an accepting run from the state f on the word ω under the environment B. We proceed by induction on the length $|\omega|$.

- **Basis** $|\omega| = 0$: Then $\omega = \epsilon$. Since M does not have any accepting run from the state f on the word ϵ under the environment B, we get that $f \notin A$. Since $A' = Q \cup \{q_f\}\backslash A$, then $f \in A'$, we get that $\overline{M}$ has an accepting run from the state f on the word ϵ under the environment B.
- **Step** $|\omega| \geq 1$: Let $\gamma \in \Gamma, u \in \Gamma^*$ such that $\omega = \gamma u$. The proof depends on the case whether M has a transition rule either of the form $f \xrightarrow{x} \{q_1, \ldots, q_m\} \in \delta$, or $f \xrightarrow{\neg x} \{q_1, \ldots, q_m\} \in \delta$, or $f \xrightarrow{\gamma} S_i \in \delta$, or does not have any transition.

  - **Case 1**: $f \xrightarrow{x} \{q_1, \ldots, q_m\} \in \delta$, then we get that M does not have any other transition rule from the state f due to the assumption. The proof depends on the case whether $B(x) = \gamma$.
    - $B(x) = \gamma$: Then the run of M from the state f will move to the states $q_1, \ldots, q_m$. Since M does

not have any accepting run from the state f on the word ω under the environment B, we obtain that there are some states $q_i \in \{q_1, \ldots, q_m\}$ such that M does not have any accepting run from the state $q_i \in Q$ on the word u under the environment B. By applying the induction hypothesis, we obtain that $\overline{M}$ has an accepting run from these states q_i on the word u under the environment B. Since $f \xrightarrow{x} \{q_1, \ldots, q_m\} \in \delta$, we get that $f \xrightarrow{x} q_i \in \delta'$ for every $1 \le i \le m$. Thus, $\overline{M}$ has an accepting run from the state f on the word γu under the environment B.

- $B(x) \neq \gamma$: Since $f \xrightarrow{x} \{q_1, \ldots, q_m\} \in \delta$, we get that $f \xrightarrow{\neg x} q_f \in \delta'$. Since $q_f \xrightarrow{\gamma_1} q_f \in \delta'$ for every $\gamma_1 \in \Gamma$ and $q_f \in A'$, we obtain that $\overline{M}$ has an accepting run from the state f on the word γu under the environment B.

- **Case 2**: $f \xrightarrow{\neg x} \{q_1, \ldots, q_m\} \in \delta$, then we get that M does not have any other transition rule from the state f due to the assumption. The proof depends on the case whether $B(x) = \gamma$.

  - $B(x) \neq \gamma$: Then the run of M from the state f will move to the states $q_1, \ldots, q_m$. Since M does not have any accepting run from the state f on the word ω under the environment B, we obtain that there are some states $q_i \in \{q_1, \ldots, q_m\}$ such that M does not have any accepting run from the state $q_i \in Q$ on the word u under the environment B. By applying the induction hypothesis, we obtain that $\overline{M}$ has an accepting run from these states q_i on the word u under the environment B. Since $f \xrightarrow{\neg x} \{q_1, \ldots, q_m\} \in \delta$, we get that $f \xrightarrow{\neg x} q_i \in \delta'$ for every $1 \le i \le m$. Thus, $\overline{M}$ has an accepting run from the state f on the word γu under the environment B.
  - $B(x) = \gamma$: Since $f \xrightarrow{\neg x} \{q_1, \ldots, q_m\} \in \delta$, we get that $f \xrightarrow{x} q_f \in \delta'$. Since $q_f \xrightarrow{\gamma_1} q_f \in \delta'$ for every $\gamma_1 \in \Gamma$ and $q_f \in A'$, we obtain that $\overline{M}$ has an accepting run from the state f on the word γu under the environment B.

- **Case 3**: $f \xrightarrow{\gamma} S_i \in \delta$ for every $1 \le i \le m$. Then, the run of M from the state f can move to one state S_i of the states $\{S_1, \ldots, S_m\}$. Since M does not have any accepting run from the state f on the word ω under the environment B, we obtain that for every $1 \le i \le m$, there exists a state $q_i \in S_i$ such that M does not have any accepting run from the state q_i on the word u under the environment B. By applying the induction hypothesis, we get that $\overline{M}$ has an accepting run from these states q_i on the word u under the environment B for every $1 \le i \le m$.
  Since $f \xrightarrow{\gamma} S_i \in \delta$ for every $1 \le i \le m$, we get that $f \xrightarrow{\gamma} \{q_1, \ldots, q_m\} \in \delta'$. $\overline{M}$ has an accepting run from the state f on the word γu under the environment B.
- **Case 4**: There is no transition in the form $f \xrightarrow{\gamma} S \in \delta$, or $f \xrightarrow{x} S \in \delta$ or $f \xrightarrow{\neg x} S \in \delta$: then, we get that $f \xrightarrow{\gamma} q_f \in \delta'$ and $q_f \xrightarrow{\gamma_1} q_f \in \delta'$ for every $\gamma_1 \in \Gamma$. Since $q_f \in A'$, we obtain that $\overline{M}$ has an accepting run from the state f on the word γu under the environment B.

Intersection. Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, let $M^1 = (Q^1, \Gamma, \delta^1, q_0^1, A^1)$ and $M^2 = (Q^2, \Gamma, \delta^2, q_0^2, A^2)$ be two VAs, we construct a VA M such that $L(M) = L(M_1) \cap L(M_2)$.

Since VA are closed under complementation, let $\overline{M^1}$ and $\overline{M^2}$ be two VAs such that $L(\overline{M^1}) = (P \times \Gamma^*) \times \mathcal{B} \backslash L(M^1)$ and $L(\overline{M^2}) = (P \times \Gamma^*) \times \mathcal{B} \backslash L(M^2)$.

Since VA is closed under union, we construct a VA M_3 such that $L(M_3) = L(\overline{M^1}) \cup L(\overline{M^2})$. Then, we can compute a VA $\overline{M^3}$ such that $L(\overline{M^3}) = P \times \Gamma^* \times \mathcal{B} \backslash L(M^3)$.

According to the above constructions, we obtain that $L(\overline{M^3}) = P \times \Gamma^* \times \mathcal{B} \backslash L(M^3) = P \times \Gamma^* \times \mathcal{B} \backslash (L(\overline{M^1}) \cup L(\overline{M^2})) = L(M_1) \cap L(M_2)$. □

A.2 Proof of Theorem 2

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

Proof To construct a VA M s.t. $L(M) = L(e)$. We first construct a Variable Automaton with ϵ-transitions (ϵ-VA) M_ϵ where transition rules are in the form of:

- $p \xrightarrow{\alpha} q$ s.t. $\alpha \in \mathcal{X} \cup \Gamma$, $p, q \in Q$; or
- $p \xrightarrow{\epsilon} q$ which can be fired without consuming any input symbol.

Then, we can translate the ϵ-VA M_ϵ into an equivalent VA M by performing ϵ-transitions elimination as usual.

Given a regular expression $e \in \mathcal{R}$, we can construct an ϵ-VA M_ϵ by induction on the structure of e.

- $e = \emptyset$: Let $M_\epsilon = (Q, \Gamma, \delta, q_0, A)$ where $Q = \{q_0, f\}$, $A = \{f\}$, $\delta = \emptyset$. Then, we obtain that $L(M_\epsilon) = \emptyset = L(\emptyset)$.
- $e = \epsilon$: Let $M_\epsilon = (Q, \Gamma, \delta, q_0, A)$ where $Q = \{q_0, f\}$, $A = \{f\}$, $\delta = \{q_0 \xrightarrow{\epsilon} f\}$. Then, we obtain that $L(M_\epsilon) = \{(\langle p, \epsilon \rangle, B) \mid \forall p \in P, B \in \mathcal{B}\} = L(\epsilon)$.
- $e = a \in \mathcal{X}$: Let $M_\epsilon = (Q, \Gamma, \delta, q_0, A)$ where $Q = \{q_0, f\}$, $A = \{f\}$, $\delta = \{q_0 \xrightarrow{a} f\}$. Then, we obtain

that $L(M_\epsilon) = \{(\langle p, \gamma\rangle, B) \mid \forall p \in P, \gamma \in \Gamma, B \in \mathcal{B} : B(a) = \gamma\} = L(a)$.

- $e = a \in \Gamma$: Let $M_\epsilon = (Q, \Gamma, \delta, q_0, A)$ where $Q = \{q_0, f\}$, $A = \{f\}$, $\delta = \{q_0 \xrightarrow{a} f\}$. Then, we obtain that $L(M_\epsilon) = \{(\langle p, a\rangle, B) \mid \forall p \in P, B \in \mathcal{B}\} = L(a)$.
- $e = e_1 + e_2$: By applying the induction hypothesis, there exist $M_\epsilon^1 = (Q^1, \Gamma, \delta^1, q_0^1, \{f^1\})$ and $M_\epsilon^2 = (Q^2, \Gamma, \delta^2, q_0^2, \{f^2\})$ such that $L(M_\epsilon^1) = L(e_1)$ and $L(M_\epsilon^2) = L(e_2)$. Let $M_\epsilon = (Q, \Gamma, \delta, q_0, \{f\})$, where $Q = Q^1 \cup Q^2 \cup \{q_0, f\}$, $\delta = \delta^1 \cup \delta^2 \cup \delta'$, δ' consists of the following transitions:
  - $q_0 \xrightarrow{\epsilon} q_0^1 \in \delta'$,
  - $q_0 \xrightarrow{\epsilon} q_0^2 \in \delta'$,
  - $f^1 \xrightarrow{\epsilon} f \in \delta'$,
  - $f^2 \xrightarrow{\epsilon} f \in \delta'$.

  For every $(\langle p, \omega\rangle, B) \in P \times \Gamma^* \times \mathcal{B}$, $(\langle p, \omega\rangle, B) \in L(M_\epsilon^1)$ or $(\langle p, \omega\rangle, B) \in L(M_\epsilon^2)$ iff $(\langle p, \epsilon\omega\epsilon\rangle, B) \in L(M_\epsilon)$. Thus, $L(M_\epsilon) = L(M_\epsilon^1) \cup L(M_\epsilon^2) = L(e_1) \cup L(e_2) = L(e)$.
- $e = e_1 \cdot e_2$: By applying the induction hypothesis, there exist $M_\epsilon^1 = (Q^1, \Gamma, \delta^1, q_0^1, \{f^1\})$ and $M_\epsilon^2 = (Q^2, \Gamma, \delta^2, q_0^2, \{f^2\})$ such that $L(M_\epsilon^1) = L(e_1)$ and $L(M_\epsilon^2) = L(e_2)$. Let $M_\epsilon = (Q, \Gamma, \delta, q_0^1, \{f^2\})$, where $Q = Q^1 \cup Q^2$, $\delta = \delta^1 \cup \delta^2 \cup \{f^1 \xrightarrow{\epsilon} q_0^2\}$.
  For every $(\langle p, \omega\rangle, B) \in P \times \Gamma^* \times \mathcal{B}$, $(\langle p, \omega_1\rangle, B) \in L(M_\epsilon^1)$ and $(\langle p, \omega_2\rangle, B) \in L(M_\epsilon^2)$ iff $(\langle p, \omega_1\epsilon\omega_2\rangle, B) \in L(M_\epsilon)$ (i.e., $(\langle p, \omega_1\omega_2\rangle, B) \in L(M_\epsilon)$). Thus, $L(M_\epsilon) = L(e)$.
- $e = e_1^*$: By applying the induction hypothesis, there exists $M_\epsilon^1 = (Q^1, \Gamma, \delta^1, q_0^1, \{f^1\})$ such that $L(M_\epsilon^1) = L(e_1)$. Let $M_\epsilon = (Q, \Gamma, \delta, q_0, \{f\})$, where $Q = Q^1 \cup \{q_0, f\}$, $\delta = \delta^1 \cup \delta'$, δ' consists of the following transitions:
  - $q_0 \xrightarrow{\epsilon} q_0^1 \in \delta'$,
  - $q_0 \xrightarrow{\epsilon} f \in \delta'$
  - $f^1 \xrightarrow{\epsilon} q_0^1 \in \delta'$,
  - $f^1 \xrightarrow{\epsilon} f \in \delta'$.

  For every $(\langle p, \omega\rangle, B) \in P \times \Gamma^* \times \mathcal{B} : (\langle p, \omega\rangle, B) \in L(M_\epsilon)$ iff $\omega \in \{u \mid (\langle p, u\rangle, B) \in L(M_\epsilon^1)\}^*$. Thus, $L(M_\epsilon) = L(e)$.

Finally, we can eliminate all the ϵ transitions from M_ϵ in the standard manner obtaining the VA M, i.e., as done for finite-state automata. □

A.3 Proof of Theorem 3

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 $B \in \mathcal{B}$, $\langle p, \omega\rangle \models_\lambda^B \varphi$ iff $\mathcal{BP}_\varphi$ has an accepting run from the configuration $\langle [(\!|p, \varphi|\!), B], \omega\rangle$.*

Proof ($\Longrightarrow$) Suppose $\langle p, \omega\rangle \models_\lambda^B \varphi$, we show that $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!|p, \varphi|\!), B], \omega\rangle$ by induction on the structure of φ.

Case $\varphi = a(x_1, \ldots, x_n) \in AP^+(\varphi)$: Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, then $\langle p, \omega\rangle \in \lambda(a(B(x_1), \ldots, B(x_n)))$. This implies that $[(\!|p, \varphi|\!), B]$ is an accepting control location.

Since $\langle (\!|p, \varphi|\!), \gamma\rangle \overset{id}{\hookrightarrow} \langle (\!|p, \varphi|\!), \gamma\rangle \in \Delta$ for every $\gamma \in \Gamma$. Thus, $\mathcal{BP}_\varphi$ has an accepting run from the configuration $\langle [(\!|p, \varphi|\!), B], \omega\rangle$, i.e., $\mathcal{BP}_\varphi$ has a run from the configuration $\langle [(\!|p, \varphi|\!), B], \omega\rangle$ which infinitely often visits some configurations with accepting control locations.

Case $\varphi = \neg a(x_1, \ldots, x_n) \in AP^+(\varphi)$: Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, then $\langle p, \omega\rangle \notin \lambda(a(B(x_1), \ldots, B(x_n)))$. This implies that $[(\!|p, \varphi|\!), B]$ is an accepting control location.

Since $\langle (\!|p, \varphi|\!), \gamma\rangle \overset{id}{\hookrightarrow} \langle (\!|p, \varphi|\!), \gamma\rangle \in \Delta$ for every $\gamma \in \Gamma$. Thus, $\mathcal{BP}_\varphi$ has an accepting run from the configuration $\langle [(\!|p, \varphi|\!), B], \omega\rangle$.

Case $\varphi = e$: Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, then $(\langle p, \omega\rangle, B) \in L(M_e)$.

Since the run of $\mathcal{BP}_\varphi$ starting from $\langle [(\!|p, \varphi|\!), B], \omega\rangle$ moves to $\langle [s_e, B], \omega\rangle$ where s_e is the initial state of the VA M_e, and the run of $\mathcal{BP}_\varphi$ starting from $\langle [s_e, B], \omega\rangle$ mimics the run of M_e on the word ω. It is sufficient to prove that:

if M_e has an accepting run from a state q on the word u under the environment B, then $\mathcal{BP}_\varphi$ has an accepting run from $\langle [q, B], u\rangle$. We proceed by applying induction on the length of u.

- **Basis** $|u| = 0$: Then $u = \epsilon$. Then we get that $q \in A_e$. Since $\langle q, \sharp\rangle \xrightarrow{id} \langle q, \sharp\rangle$ and $[q, B]$ is an accepting control location. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle [q, B], \sharp\rangle$. Note that $\sharp$ is the bottom of the stack when the stack content is empty.
- **Step** $|u| \geq 1$: Let $\gamma \in \Gamma$, $v \in \Gamma^*$ such that $u = \gamma v$. Let t be the first transition rule used by the run of M_e. The proof depends on the type of t.
  - **Case** $t = q \xrightarrow{x} \{q_1, \ldots, q_m\}$ and $x \in \mathcal{X}$. Then $B(x) = \gamma$, and M_e has an accepting run from the state q_i on the word v under the environment B for every $1 \leq i \leq m$. By applying the induction hypothesis, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle [q_i, B], v\rangle$ for every $1 \leq i \leq m$. Since $\langle q, \gamma\rangle \overset{join_\gamma^x}{\hookrightarrow} \{\langle q_1, \epsilon\rangle, \ldots, \langle q_m, \epsilon\rangle\} \in \Delta$ and the relation $join_\gamma^x$ guarantees that $B \in join_\gamma^x(B, \ldots, B)$ (Since $B(x) = \gamma$). Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle [q, B], \omega\rangle$.
  - **Case** $t = q \xrightarrow{\neg x} \{q_1, \ldots, q_m\}$ and $x \in \mathcal{X}$. Then $B(x) \neq \gamma$, and M_e has an accepting run from the state q_i on the word v under the environment B

for every $1 \leq i \leq m$. By applying the induction hypothesis, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q_i, B], v\rangle$ for every $1 \leq i \leq m$. Since $\langle q, \gamma\rangle \overset{join_\gamma^{\neg x}}{\hookrightarrow} \{\langle q_1, \epsilon\rangle, \dots, \langle q_m, \epsilon\rangle\} \in \Delta$ and the relation $join_\gamma^{\neg x}$ ensures that $B \in join_\gamma^{\neg x}(B, \dots, B)$ (Since $B(x) \neq \gamma$). Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q, B], \omega\rangle$.

- **Case** $t = q \xrightarrow{\gamma} \{q_1, \dots, q_m\}$. Then, M_e has an accepting run from state q_i on the word v under the environment B for every $1 \leq i \leq m$. By applying the induction hypothesis, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q_i, B], v\rangle$ for every $1 \leq i \leq m$. Since $\langle q, \gamma\rangle \overset{equal}{\hookrightarrow} \{\langle q_1, \epsilon\rangle, \dots, \langle q_m, \epsilon\rangle\} \in \Delta$ and the relation $equal$ ensures that $B \in equal(B, \dots, B)$. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q, B], \omega\rangle$.

Case $\varphi = \neg e$: It is similar to the case where $\varphi = e$.
Case $\varphi = \varphi_1 \wedge \varphi_2$: Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, we get that $\langle p, \omega\rangle \models_\lambda^B \varphi_1$ and $\langle p, \omega\rangle \models_\lambda^B \varphi_2$.

By applying the induction hypothesis, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi_1|\!), B], \omega\rangle$ and $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi_2|\!), B], \omega\rangle$.

Since $\langle(\!|p, \varphi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle(\!|p, \varphi_1|\!), \gamma\rangle, \langle(\!|p, \varphi_2|\!), \gamma\rangle]$ for every $\gamma \in \Gamma$ and $B \in equal(B, B)$, we get that $\langle[(\!|p, \varphi|\!), B], \omega\rangle$ is an immediate predecessor of $\{\langle[(\!|p, \varphi_1|\!), B], \omega\rangle, \langle[(\!|p, \varphi_2|\!), B], \omega\rangle\}$. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi|\!), B], \omega\rangle$.
Case $\varphi = \varphi_1 \vee \varphi_2$: Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, we get that $\langle p, \omega\rangle \models_\lambda^B \varphi_1$ or $\langle p, \omega\rangle \models_\lambda^B \varphi_2$.

By applying the induction hypothesis, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi_1|\!), B], \omega\rangle$ or $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi_2|\!), B], \omega\rangle$.

Since $\langle(\!|p, \varphi|\!), \gamma\rangle \overset{id}{\hookrightarrow} \langle(\!|p, \varphi_1|\!), \gamma\rangle$ or $\langle(\!|p, \varphi|\!), \gamma\rangle \overset{id}{\hookrightarrow} \langle(\!|p, \varphi_2|\!), \gamma\rangle$ and $B \in id(B)$, we get that $\langle[(\!|p, \varphi|\!), B], \omega\rangle$ is an immediate predecessor of $\langle[(\!|p, \varphi_1|\!), B], \omega\rangle$ and of $\langle[(\!|p, \varphi_2|\!), B], \omega\rangle$. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi|\!), B], \omega\rangle$.
Case $\varphi = \forall x \varphi_1$: Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, we get that $\langle p, \omega\rangle \models_\lambda^{B[x \leftarrow v]} \varphi_1$, for every $v \in \mathcal{D}$. Suppose $\mathcal{D} = \{c_1, \dots, c_n\}$. By applying the induction hypothesis, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi_1|\!), B[x \leftarrow c_i]], \omega\rangle$ for every $1 \leq i \leq n$. Since $\langle(\!|p, \varphi|\!), \gamma\rangle \overset{meet_\mathcal{D}^x}{\hookrightarrow} [\langle(\!|p, \varphi_1|\!), \gamma\rangle, \dots, \langle(\!|p, \varphi_1|\!), \gamma\rangle]$ for every $\gamma \in \Gamma$ and the relation $meet_\mathcal{D}^x$ ensures that $B \in meet_\mathcal{D}^x(B[x \leftarrow c_1], \dots, B[x \leftarrow c_n])$, we get that $\langle[(\!|p, \varphi|\!), B], \omega\rangle$ is an immediate predecessor of $\{\langle[(\!|p, \varphi_1|\!), B[x \leftarrow c_1]], \omega\rangle, \dots, \langle[(\!|p, \varphi_1|\!), B[x \leftarrow c_n]], \omega\rangle\}$. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi|\!), B], \omega\rangle$.
Case $\varphi = \exists x \varphi_1$: Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, there exists a $v \in \mathcal{D}$ such that $\langle p, \omega\rangle \models_\lambda^{B[x \leftarrow v]} \varphi_1$.

By applying the induction hypothesis, we get that $\mathcal{BP}$ has an accepting run from $\langle[(\!|p, \varphi_1|\!), B[x \leftarrow v]], \omega\rangle$.

Since for every $\gamma \in \Gamma$ $\langle(\!|p, \varphi|\!), \gamma\rangle \overset{meet_{\{v\}}^x}{\hookrightarrow} \langle(\!|p, \varphi_1|\!), \gamma\rangle$ and the relation $meet_{\{v\}}^x$ ensures that $B \in meet_{\{v\}}^x(B)$, we get that $\langle[(\!|p, \varphi|\!), B], \omega\rangle$ is an immediate predecessor of $\langle[(\!|p, \varphi_1|\!), B[x \leftarrow v]], \omega\rangle$. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi|\!)), B], \omega\rangle$.
Case $\varphi = \mathbf{EX}\varphi_1$: Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, then there exists an immediate successor $\langle p', \omega'\rangle$ of $\langle p, \omega\rangle$ such that $\langle p', \omega'\rangle \models_\lambda^B \varphi_1$ and $\langle p, \omega\rangle \Longrightarrow_\mathcal{P} \langle p', \omega'\rangle$.

By applying the induction hypothesis, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p', \varphi_1|\!), B], \omega'\rangle$.

Since $\langle(\!|p, \varphi|\!), \gamma\rangle \overset{id}{\hookrightarrow} \langle(\!|p', \varphi|\!), \omega\rangle$ and $B \in id(B)$, we get that $\langle[(\!|p, \varphi|\!), B], \omega\rangle$ is an immediate predecessor of $\langle[(\!|p', \varphi_1|\!), B], \omega'\rangle$. Hence, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi|\!), B], \omega\rangle$.
Case $\varphi = \mathbf{AX}\varphi_1$: Let n be the number of immediate successors of $\langle p, \omega\rangle$. Since $\langle p, \omega\rangle \models_\lambda^B \varphi$, then for each immediate successors $\langle p_i, \omega_i\rangle$ of $\langle p, \omega\rangle$: $\langle p_i, \omega_i\rangle \models_\lambda^B \varphi$ and $\langle p, \omega\rangle \Longrightarrow_\mathcal{P} \langle p_i, \omega_i\rangle$, for every $1 \leq i \leq n$.

By applying the induction hypothesis, we obtain that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_i, \varphi_1|\!), B], \omega_i\rangle$ for each $1 \leq i \leq n$.

Since $\langle(\!|p, \varphi|\!), \gamma\rangle \overset{equal}{\hookrightarrow} [\langle(\!|p_1, \varphi_1|\!), \omega_1\rangle, \dots, \langle(\!|p_n, \varphi_1|\!), \omega_n\rangle]$ and $B \in equal(B, \dots, B)$, we get that $\langle[(\!|p, \varphi|\!), B], \omega\rangle$ is an immediate predecessor of $\{\langle[(\!|p_1, \varphi_1|\!), B], \omega_1\rangle, \dots, \langle[(\!|p_n, \varphi_1|\!), B], \omega_n\rangle\}$. Hence, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi|\!), B], \omega\rangle$.
Case $\varphi = \mathbf{E}[\varphi_1 \mathbf{U} \varphi_2]$: Since $\langle p, \omega\rangle \models_\lambda^B \mathbf{E}[\varphi_1 \mathbf{U} \varphi_2]$, then there exists a path $\langle p_0, \omega_0\rangle\langle p_1, \omega_1\rangle, \langle p_2, \omega_2\rangle \cdots$ from $\langle p, \omega\rangle$ such that $\exists i \geq 0$, $\langle p_i, \omega_i\rangle \models_\lambda^B \varphi_2$ and $\forall 0 \leq j < i : \langle p_j, \omega_j\rangle \models_\lambda^B \varphi_1$. Since $\langle p_i, \omega_i\rangle \models_\lambda^B \varphi_2$ and $\langle p_j, \omega_j\rangle \models_\lambda^B \varphi_1$ for every $0 \leq j < i$. By applying the induction hypothesis, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_i, \varphi_2|\!), B], \omega_i\rangle$ and for every $0 \leq j < i$, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_j, \varphi_1|\!), B], \omega_j\rangle$.

Since $\langle(\!|p_i, \varphi|\!), \gamma\rangle \overset{id}{\hookrightarrow} \langle(\!|p_i, \varphi_2|\!), \gamma\rangle$ and $B \in id(B)$, we obtain that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_i, \varphi|\!), B], \omega_i\rangle$.

If $i = 0$, then $\langle[(\!|p, \varphi|\!), B], \omega\rangle = \langle[(\!|p_i, \varphi|\!), B], \omega_i\rangle$, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p, \varphi|\!), B], \omega\rangle$.

Otherwise $i > 0$, we prove that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_j, \varphi|\!), B], \omega_j\rangle$ by applying induction on $l = i - j$. (Note that $\langle[(\!|p_0, \varphi|\!), B], \omega_0\rangle = \langle[(\!|p, \varphi|\!), B], \omega\rangle$.)

- **Basis**. $l = 1$. Then there exists $\langle p_j, \omega_j\rangle \Longrightarrow_\mathcal{P} \langle p_i, \omega_i\rangle$. According to the product of $\mathcal{BP}_\varphi$, We get that $\langle[(\!|p_j, \varphi|\!), B], \omega_j\rangle$ is an immediate predecessor of $\{\langle[(\!|p_j, \varphi_1|\!), B], \omega_j\rangle, \langle[(\!|p_i, \varphi|\!), B], \omega_i\rangle\}$. This implies that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_j, \varphi|\!), B], \omega_j\rangle$.
- **Step**. $l > 1$. Then there exists $\langle p_{j+1}, \omega_{j+1}\rangle$ such that $\langle p_j, \omega_j\rangle \Longrightarrow_\mathcal{P} \langle p_{j+1}, \omega_{j+1}\rangle \Longrightarrow_\mathcal{P} \langle p_i, \omega_i\rangle$. By the

induction hypothesis (induction on l), we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_{j+1},\varphi|\!), B], \omega_{j+1}\rangle$.
Since $\langle p_j, \omega_j\rangle \models^B_\lambda \varphi_1$, by applying the induction hypothesis (induction on structure of φ), we obtain that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_j,\varphi_1|\!), B], \omega_j\rangle$. Since $\langle[(\!|p_j,\varphi|\!), B], \omega_j\rangle$ is an immediate predecessor of $\{\langle[(\!|p_j,\varphi_1|\!), B], \omega_j\rangle, \langle[(\!|p_{j+1},\varphi|\!), B], \omega_{j+1}\rangle\}$, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$.

Case $\varphi = \mathbf{A}[\varphi_1\mathbf{U}\varphi_2]$: We can prove that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$ as done for to $\varphi = \mathbf{E}[\varphi_1\mathbf{U}\varphi_2]$.
Case $\varphi = \mathbf{E}[\varphi_1\mathbf{R}\varphi_2]$: Since $\langle p, \omega\rangle) \models^B_\lambda \mathbf{E}[\varphi_1\mathbf{R}\varphi_2]$, then there exists a path $\rho = \langle p_0, \omega_0\rangle\langle p_1\omega_1\rangle, \langle p_2, \omega_2\rangle\cdots$ from $\langle p, \omega\rangle$ such that

1. $\forall i \geq 0,\ \langle p_i, \omega_i\rangle \models^B_\lambda \varphi_2$,
2. or there exists $i \geq 0$ such that $\langle p_i, \omega_i\rangle \models^B_\lambda \varphi_1$ and $\forall 0 \leq j \leq i,\ \langle p_i, \omega_i\rangle \models^B_\lambda \varphi_2$

– First we consider case (2), it can be proved that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$ by applying the induction on $i - j$ similar to the case where $\varphi = \mathbf{E}[\varphi_1\mathbf{U}\varphi_2]$.
– Considering the case (1), let us prove that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$. According to the semantics of SCTPL, $\mathcal{P}$ has an infinite path $r = \langle p_0, \omega_0\rangle, \langle p_1\,\omega_1\rangle, \langle p_2, \omega_2\rangle, \ldots, \langle p_i, \omega_i\rangle, \cdots$ such that $\langle p_i, \omega_i\rangle \models^B_\lambda \varphi_2$ for all $i \geq 0$. Since the number of control locations and stack alphabet of $\mathcal{P}$ is finite and the path r is infinite, then there exists a configuration $\langle p_m, \gamma u\rangle$ such that $\omega_m = \gamma u$, $\langle p_0, \omega_0\rangle \Longrightarrow_{\mathcal{P}} \langle p_m, \gamma u\rangle$ and $\langle p_m, \gamma\rangle \Longrightarrow_{\mathcal{P}} \langle p_m, \gamma v\rangle$ (Proposition 3 of [7]). This implies that $\langle p_m, \gamma u\rangle \Longrightarrow_{\mathcal{P}} \langle p_m, \gamma v u\rangle$. Let $\langle p_n, \omega_n\rangle$ be the first configuration such that $\langle p_n, \omega_n\rangle = \langle p_m, \gamma v u\rangle$.
Since for each configuration $\langle p_k, \omega_k\rangle$ in the run $\langle p_m, \gamma u\rangle \Longrightarrow_{\mathcal{P}} \langle p_m, \gamma v u\rangle$: $\langle[(\!|p_k,\varphi|\!), B], \omega_k\rangle$ is an immediate predecessor of $\{\langle[(\!|p_k,\varphi_2|\!), B], \omega_k\rangle, \langle[(\!|p_{k+1},\varphi|\!), B], \omega_{k+1}\rangle\}$. According to the definition of the reachability relation of $\mathcal{BP}_\varphi$, we obtain that
$\langle[(\!|p_m,\varphi|\!), B], \gamma u\rangle \in Pre^+(\{\langle[(\!|p_m,\varphi|\!), B], \gamma v u\rangle, \langle[(\!|p_{m+0},\varphi_2|\!), B], \omega_{m+0}\rangle, \ldots, \langle[(\!|p_n,\varphi_2|\!), B], \omega_n\rangle\})$.
Since $\langle p_i, \omega_i\rangle \models^B_\lambda \varphi_2$ for every $i \geq 0$, by applying the induction hypothesis, we obtain that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p_i,\varphi_2|\!), B], \omega_i\rangle$.
Since for each $i \geq 0$ $[(\!|p_i,\varphi|\!), B] \in F$ which implies that $[(\!|p_i,\varphi|\!), B]$ is an accepting control location, then $\mathcal{BP}_\varphi$ has a run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$ such that each path will infinitely often visit some configurations $\langle[(\!|p_i,\varphi|\!), B], \omega_i\rangle$ with accepting control locations. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$.

Case $\varphi = \mathbf{A}[\varphi_1\mathbf{R}\varphi_2]$: We can prove that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$ as done for $\varphi = \mathbf{E}[\varphi_1\mathbf{R}\varphi_2]$.

($\Longleftarrow$) $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$, we show that $\langle p, \omega\rangle \models^B_\lambda \varphi$ by applying induction on the structure of φ.
Case $\varphi = a(x_1, \ldots, x_n) \in AP^+(\varphi)$: Since $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$ and $\langle(\!|p,\varphi|\!), \omega\rangle \overset{id}{\hookrightarrow} \langle p_\varphi, \omega\rangle$, $\langle p, \omega\rangle \in \lambda(a(B(x_1), .., B(x_n))$. Thus, $\langle p, \omega\rangle \models^B_\lambda \varphi$.
Case $\varphi = \neg a(x_1, \ldots, x_n) \in AP^-(\varphi)$: Since $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$ and $\langle(\!|p,\varphi|\!), \omega\rangle \overset{id}{\hookrightarrow} \langle p_\varphi, \omega\rangle$, this implies that $\langle p, \omega\rangle \notin \lambda(a(B(x_1), \ldots, B(x_n)))$. Thus, $\langle p, \omega\rangle \models^B_\lambda \varphi$.
Case $\varphi = e$: Since $\mathcal{BP}_\varphi$ has an accepting run from $\langle[(\!|p,\varphi|\!), B], \omega\rangle$ and $\langle(\!|p,\varphi|\!), \omega\rangle \overset{id}{\hookrightarrow} \langle s_e, \omega\rangle$, we get that $\mathcal{BP}_\varphi$ has an accepting run from $\langle[s_e, B], \omega\rangle$. Since the run of $\mathcal{BP}_\varphi$ from the configuration $\langle[s_e, B], \omega\rangle$ mimics the run of M_e. It is sufficient to prove that if $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q, B], u\rangle$ for every $q \in Q, u \in \Gamma^*$, then M_e has an accepting run from the state q on the word u under B. We proceed by induction on the length $|u|$.

– **Basis** $|u| = 0$: Then $u = \epsilon$. Since $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q, B], \sharp\rangle$, $\langle q, \sharp\rangle \xrightarrow{id} \langle q, \sharp\rangle \in \Delta$ and $[q, B]$ is accepting, we get that $q \in A_e$. M_e has an accepting run from the state q on the word u under B. Note that $\sharp$ is the bottom of the stack (i.e., the stack content is ϵ).
– **Step** $|u| \geq 1$: Let $\gamma \in \Gamma$, $v \in \Gamma^*$ such that $u = \gamma v$. Let t be the first transition rule used by the run of $\mathcal{BP}_\varphi$. The proof depends on the type of t.

  – **Case** $t = \langle q, \gamma\rangle \overset{join^x_\gamma}{\hookrightarrow} \{\langle q_1, \epsilon\rangle, \ldots, \langle q_m, \epsilon\rangle\} \in \Delta$, then $q \xrightarrow{x} \{q_1, \ldots, q_m\}$ and $x \in \mathcal{X}$. The relation $join^x_\gamma$ ensures that $B(x) = \gamma$.
  $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q, B], u\rangle$ for every $q \in Q, u \in \Gamma^*$, then $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q_i, B], v\rangle$ for every $1 \leq i \leq m$. By applying the induction hypothesis, we obtain that M_e has an accepting run from state q_i on the word v under B for every $1 \leq i \leq m$.
  Since $q \xrightarrow{x} \{q_1, \ldots, q_m\}$, we get that M_e has an accepting run from the state q on the word u under B.
  – **Case** $t = \langle q, \gamma\rangle \overset{join^{\neg x}_\gamma}{\hookrightarrow} \{\langle q_1, \epsilon\rangle, \ldots, \langle q_m, \epsilon\rangle\} \in \Delta$, then $q \xrightarrow{\neg x} \{q_1, \ldots, q_m\}$ and $x \in \mathcal{X}$. The relation $join^{\neg x}_\gamma$ ensures that $B(x) \neq \gamma$.
  $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q, B], u\rangle$ for every $q \in Q, u \in \Gamma^*$, then $\mathcal{BP}_\varphi$ has an accepting run from $\langle[q_i, B], v\rangle$ for every $1 \leq i \leq m$. By applying the induction hypothesis, we obtain that M_e has an accepting run from state q_i on the word v under B for every $1 \leq i \leq m$.

Since $q \xrightarrow{\neg x} \{q_1, \ldots, q_m\}$, we get that M_e has an accepting run from the state q on the word u under B.

- **Case** $t = \langle q, \gamma \rangle \overset{equal}{\hookrightarrow} \{\langle q_1, \epsilon \rangle, \ldots, \langle q_m, \epsilon \rangle\} \in \Delta$, then $q \xrightarrow{\gamma} \{q_1, \ldots, q_m\}$.
  $\mathcal{BP}_\varphi$ has an accepting run from $\langle [q, B], u \rangle$ for every $q \in Q, u \in \Gamma^*$, then $\mathcal{BP}_\varphi$ has an accepting run from $\langle [q_i, B], v \rangle$ for every $1 \leq i \leq m$. By applying the induction hypothesis, we obtain that M_e has an accepting run from state q_i on the word v under B for every $1 \leq i \leq m$.
  Since $q \xrightarrow{\gamma} \{q_1, \ldots, q_m\}$, we get that M_e has an accepting run from the state q on the word u under B.

Case $\varphi = \neg e$: This case is similar to the case where $\varphi = e$.

Case $\varphi = \varphi_1 \wedge \varphi_2$: Since $\langle (\!| p, \varphi |\!), \omega \rangle \overset{equal}{\hookrightarrow} [\langle (\!| p, \varphi_1 |\!), \omega \rangle, \langle (\!| p, \varphi_2 |\!), \omega \rangle]$ and $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi |\!), B], \omega \rangle$, we obtain that $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi_1 |\!), B], \omega \rangle$ and from $\langle [(\!| p, \varphi_2 |\!), B], \omega \rangle$.

By applying the induction hypothesis, we obtain that $\langle p, \omega \rangle \models_\lambda^B \varphi_1$ and $\langle p, \omega \rangle \models_\lambda^B \varphi_2$. These imply that $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Case $\varphi = \varphi_1 \vee \varphi_2$: Since $\langle (\!| p, \varphi |\!), \omega \rangle \overset{id}{\hookrightarrow} \langle (\!| p, \varphi_1 |\!), \omega \rangle$ and $\langle (\!| p, \varphi |\!), \omega \rangle \overset{id}{\hookrightarrow} \langle (\!| p, \varphi_2 |\!), \omega \rangle$, and $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi |\!), B], \omega \rangle$, we obtain that $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi_1 |\!), B], \omega \rangle$ or $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi_2 |\!), B], \omega \rangle$. By applying the induction hypothesis, we have that $\langle p, \omega \rangle \models_\lambda^B \varphi_1$ or $\langle p, \omega \rangle \models_\lambda^B \varphi_2$. These imply that $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Case $\varphi = \forall x \varphi_1$: Let $\mathcal{D} = \{c_1, \ldots, c_n\}$. Since $\langle (\!| p, \varphi |\!), \omega \rangle \overset{meet_\mathcal{D}^x}{\hookrightarrow} [\langle (\!| p, \varphi_1 |\!), \omega \rangle, \ldots, \langle (\!| p, \varphi_1 |\!), \omega \rangle]$, the relation $meet_\mathcal{D}^x$ implies that the configurations $[\langle [(\!| p, \varphi_1 |\!), B[x \longleftarrow c_1]], \omega \rangle, \ldots, \langle [(\!| p, \varphi_1 |\!), B[x \longleftarrow c_n]], \omega \rangle]$ are the children of the configuration $\langle [(\!| p, \varphi |\!), B], \omega \rangle$ in the accepting run.

Since $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi |\!), B], \omega \rangle$, we obtain that $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi_1 |\!), B[x \longleftarrow c_i]], \omega \rangle$ for every $1 \leq i \leq n$. By applying the induction hypothesis, we get that $\langle p, \omega \rangle \models_\lambda^{B[x \longleftarrow c_i]} \varphi_1$ for every $1 \leq i \leq n$. Thus, $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Case $\varphi = \exists x \varphi_1$: Let $\mathcal{D} = \{c_1, \ldots, c_n\}$. Since $\langle (\!| p, \varphi |\!), \omega \rangle \overset{meet_{\{c_i\}}^x}{\hookrightarrow} \langle (\!| p, \varphi_1 |\!), \omega \rangle$ for every $1 \leq i \leq n$, the relation $meet_{\{c_i\}}^x$ implies that for every $1 \leq i \leq n$, the configuration $\langle [(\!| p, \varphi_1 |\!), B[x \longleftarrow c_i]], \omega \rangle$ can be the child of the configuration $\langle [(\!| p, \varphi_1 |\!), B], \omega \rangle$ in the accepting run.

Since $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi |\!), B], \omega \rangle$, we obtain that there exists $i : 1 \leq i \leq n$ such that $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p, \varphi_1 |\!), B[x \longleftarrow c_i]], \omega \rangle$. By applying the induction hypothesis, we get that $\langle p, \omega \rangle \models_\lambda^{B[x \longleftarrow c_i]} \varphi_1$. Hence, $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Case $\varphi = \mathbf{EX}\varphi_1$: Then, there exists an immediate successor $\langle [(\!| p', \varphi_1 |\!), B], \omega' \rangle$ of $\langle [(\!| p, \varphi |\!), B], \omega \rangle$ such that $\langle [(\!| p', \varphi_1 |\!), B], \omega' \rangle$ is a child of $\langle [(\!| p, \varphi |\!), B], \omega \rangle$ in the accepting run. Then $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p', \varphi_1 |\!), B], \omega' \rangle$.

By applying the induction hypothesis, $\langle p', \omega' \rangle \models_\lambda^B \varphi_1$. Thus, we obtain that $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Case $\varphi = \mathbf{AX}\varphi_1$: Then, the immediate successors $\{\langle [(\!| p_1, \varphi_1 |\!), B], \omega_1 \rangle, \cdots, \langle [(\!| p_n, \varphi_1 |\!), B], \omega_n \rangle$ of $\langle [(\!| p, \varphi |\!), B], \omega \rangle$ are the children of the configuration $\langle [(\!| p, \varphi |\!), B], \omega \rangle$ in the accepting run. Then $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p_i, \varphi_1 |\!), B], \omega_i \rangle$, for each $1 \leq i \leq n$.

By applying the induction hypothesis, $\langle p_i, \omega_i \rangle \models_\lambda^B \varphi_1$, for each $1 \leq i \leq n$. Thus, we obtain that $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Case $\varphi = \mathbf{E}[\varphi_1 \mathbf{U} \varphi_2]$: Let ρ be the accepting run from $\langle [(\!| p, \varphi |\!), B], \omega \rangle$, then, each configuration $\langle [(\!| p_i, \varphi |\!), B], \omega_i \rangle$ in ρ at most have two children $\langle [(\!| p_i, \varphi_1 |\!), B], \omega_i \rangle$ and $\langle [(\!| p_{i+1}, \varphi |\!), B], \omega_{i+1} \rangle$ or has only one child $\langle [(\!| p_i, \varphi_2 |\!), B], \omega_i \rangle$.

Since ρ is an accepting run, there exists a configuration $\langle [(\!| p_n, \varphi |\!), B], \omega_n \rangle$ in ρ, such that $\langle [(\!| p_n, \varphi |\!), B], \omega_n \rangle$ has only one child $\langle [(\!| p_n, \varphi_2 |\!), B], \omega_n \rangle$. Let $\langle [(\!| p_0, \varphi |\!), B], \omega_0 \rangle, \ldots, \langle [(\!| p_n, \varphi |\!), B], \omega_n \rangle, \cdots$ be a path of ρ. Then, $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p_i, \varphi_1 |\!), B], \omega_i \rangle$ for each $0 \leq i < n$, and $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p_n, \varphi_2 |\!), B], \omega_n \rangle$.

By applying the induction hypothesis, we obtain that $\langle p_n, \omega_n \rangle \models_\lambda^B \varphi_2$ and $\langle p_i, \omega_i \rangle \models_\lambda^B \varphi_1$ for each $0 \leq i < n$. Thus, $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Case $\varphi = \mathbf{A}[\varphi_1 \mathbf{U} \varphi_2]$: This case is similar to the case where $\varphi = \mathbf{E}[\varphi_1 \mathbf{U} \varphi_2]$.

Case $\varphi = \mathbf{E}[\varphi_1 \mathbf{R} \varphi_2]$: Let ρ be the accepting run from $\langle [(\!| p, \varphi, B |\!)], \omega \rangle$, then, each configuration $\langle [(\!| p_i, \varphi |\!), B], \omega_i \rangle$ in ρ has two children:

1. either $\langle [(\!| p_i, \varphi_1 |\!), B], \omega_i \rangle$ and $\langle [(\!| p_i, \varphi_2 |\!), B], \omega_i \rangle$
2. or $\langle [(\!| p_i, \varphi_2 |\!), B], \omega_i \rangle$ and $\langle [(\!| p_{i+1}, \varphi |\!), B], \omega_{i+1} \rangle$

1. First we consider Item (1). Since $[(\!| p_i, \varphi |\!), \mathcal{B}] \in F'$, we obtain that $[(\!| p_i, \varphi |\!), B]$ is an accepting control locations, then every configuration $\langle [(\!| p_i, \varphi |\!), B], \omega_i \rangle$ in ρ has two children $\langle [(\!| p_i, \varphi_2 |\!), B], \omega_i \rangle$ and $\langle [(\!| p_{i+1}, \varphi |\!), B], \omega_i \rangle$ for every $i \geq 0$. By applying the induction hypothesis to $\langle [(\!| p_i, \varphi_2 |\!), B], \omega_i \rangle$, we get that $\langle p_i, \omega_i \rangle \models_\lambda^B \varphi_2$ for every $i \geq 0$. We obtain that $\langle p, \omega \rangle \models_\lambda^B \varphi$
2. Let us consider Item (2). There is a configuration $\langle [(\!| p_n, \varphi |\!), B], \omega_n \rangle$ in ρ such that its two children are $\langle [(\!| p_n, \varphi_1 |\!), B], \omega_n \rangle$ and $\langle [(\!| p_n, \varphi_2 |\!), B], \omega_n \rangle$, each configuration $\langle [(\!| p_i, \varphi |\!), B], \omega_i \rangle$ from $\langle [(\!| p_0, \varphi |\!), B], \omega_0 \rangle$ to $\langle [(\!| p_n, \varphi |\!), B], \omega_n \rangle$ has children $\langle [(\!| p_i, \varphi_2 |\!), B], \omega_i \rangle$ and $\langle [(\!| p_{i+1}, \varphi |\!), B], \omega_i \rangle$. $\mathcal{BP}_\varphi$ has an accepting run from $\langle [(\!| p_n, \varphi_1 |\!), B], \omega_n \rangle$ and from $\langle [(\!| p_i, \varphi_2 |\!), B], \omega_i \rangle$ for $0 \leq i \leq n$.
   By applying the induction hypothesis, $\langle p_n, \omega_n \rangle \models_\lambda^B \varphi_1$ and $\langle p_i, \omega_i \rangle \models_\lambda^B \varphi_2$ for each $0 \leq i \leq n$. Thus, $\langle p, \omega \rangle \models_\lambda^B \varphi$.

Case $\varphi = \mathbf{A}[\varphi_1 \mathbf{R} \varphi_2]$: This case is similar to the case where $\varphi = \mathbf{E}[\varphi_1 \mathbf{R} \varphi_2]$. □

References

1. Avast. Free avast antivirus. http://www.avast.com. Version 6.0.1367
2. Avira. http://www.avira.com. Version 12.0.0.849
3. Balakrishnan, G., Gruian, R., Reps, T.W., Teitelbaum, T.: CodeSurfer/x86-a platform for analyzing x86 executables. In: CC, pp. 250–254 (2005)
4. Balakrishnan, G., Reps, T.W., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S.H., Chen, C.-H., Teitelbaum, T.: Model checking x86 executables with CodeSurfer/x86 and WPDS++. In: CAV, pp. 158–163 (2005)
5. Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. In: Symposium on Requirements Engineering for Information Security, pp. 1–8 (2001)
6. Bonfante, G., Kaczmarek, M., Marion, J.-Y.: Architecture of a morphological malware detector. J. Comput. Virol. **5**, 263–270 (2009)
7. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model checking. In: CONCUR'97. LNCS 1243 (1997)
8. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Computer Aided Verification (2011)
9. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
10. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th USENIX Security, Symposium, pp. 169–186 (2003)
11. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: ISEC, pp. 5–14 (2008)
12. Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: IEEE Symposium on Security and Privacy, pp. 32–46 (2005)
13. Eric, S.: 10 most destructive computer worms and viruses ever. http://wildammo.com/2010/10/12/10-most-destructive-computer-worms-and-viruses-ever (2010)
14. Esparza, J., Kucera, A., Schwoon, S.: Model-checking LTL with regular valuations for pushdown systems. In: TACS, pp. 316–339 (2001)
15. Esparza, J., Kucera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Inf. Comput. **186**(2), 355–376 (2003)
16. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV'01, pp. 324–336 (2001)
17. Gostev, A.: Kaspersky security bulletin, malware evolution 2010. http://www.securelist.com/en/analysis/204792161/Kaspersky_Security_Bulletin_Malware_Evolution_2010. Kaspersky Lab ZAO (2011)
18. Heavens, V. http://vx.netlux.org
19. Hex-Rays. IDAPro (2011)
20. Holzer, A., Kinder, J., Veith, H.: Using verification technology to specify and detect malware. In: EUROCAST, pp. 497–504 (2007)
21. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: DIMVA, pp. 174–187 (2005)
22. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Proactive detection of computer worms using model checking. IEEE Trans. Dependable Secure Comput. **7**(4), 424–438 (2010)
23. Kinder, J., Veith, H.: Jakstab: a static analysis platform for binaries. In: CAV, pp. 423–427 (2008)
24. Lakhotia, A., Boccardo, D.R., Singh, A., Manacero, A.: Context-sensitive analysis of obfuscated x86 executables. In: PEPM, pp. 131–140 (2010)
25. Lakhotia, A., Kumar, E.U., Venable, M.: A method for detecting obfuscated calls in malicious binaries. IEEE Trans. Softw. Eng. **31**(11), 955–968 (2005)
26. Qihoo 360. http://www.360.cn
27. Singh, P.K., Lakhotia, A.: Static verification of worm and virus behavior in binary executables using model checking. In: IAW, pp. 298–300 (2003)
28. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. In: CONCUR (2011)
29. Song, F., Touili, T.: Pushdown model checking for malware detection. In: TACAS, pp. 110–125 (2012)
30. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In: ATVA, pp. 141–153 (2006)
31. Uezato, Y., Minamide, Y.: Pushdown systems with stack manipulation. In: ATVA'13 (2013) (to appear)