LTL Model-Checking for Malware Detection*

Fu Song and Tayssir Touili

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

Abstract. Nowadays, malware has become a critical security threat. Traditional anti-viruses such as signature-based techniques and code emulation become insufficient and easy to get around. Thus, it is important to have efficient and robust malware detectors. In [23,21], CTL model-checking for PushDown Systems (PDSs) was shown to be a robust technique for malware detection. However, the approach of [23,21] lacks precision and runs out of memory in several cases. In this work, we show that several malware specifications could be expressed in a more precise manner using LTL instead of CTL. Moreover, LTL can express malicious behaviors that cannot be expressed in CTL. Thus, since LTL model-checking for PDSs is polynomial in the size of PDSs while CTL model-checking for PDSs is exponential, we propose to use LTL model-checking for PDSs for malware detection. Our approach consists of: (1) Modeling the binary program as a PDS. This allows to track the program's stack (needed for malware detection). (2) Introducing a new logic (SLTPL) to specify the malicious behaviors. SLTPL is an extension of LTL with variables, quantifiers, and predicates over the stack. (3) Reducing the malware detection problem to SLTPL model-checking for PDSs. We reduce this model checking problem to the emptiness problem of Symbolic Büchi PDSs. We implemented our techniques in a tool, and we applied it to detect several viruses. Our results are encouraging.

1 Introduction

Over the past decades, the landscape of malware's intent has changed. More and more sophisticated malwares have been designed for more general cyber-espionage purposes. For example, Stuxnet, Duqu and Flame are deployed for targeted attacks in countries, such as Iran, Israel, Sudan. Traditional antivirus techniques: code emulation and signature (pattern)-based techniques become insufficient. Indeed, code emulation techniques monitoring only several traces of programs in a limited time span may miss some malicious behaviors, and signature-based techniques using patterns of programs' codes to characterize malware can only detect known malwares.

Addressing these limitations, many efforts have been made [1,4,5,19,7,8,15,2]. Among them, model-checking is one of the efficient techniques for malware detection [4,19,7,8,15], as it allows to check the behavior (not the syntax) of the program without executing it. However, [4,19,7,8,15] use finite state graphs (automata) as program model that cannot accurately represent the program's stack. Being able to track the program's stack is very important for malware detection as explained in [18]. For example, malware writers obfuscate the system calls by using pushes and jumps to make malware hard to analyze, because anti-viruses usually determine malware by checking function calls to operating systems.

To overcome this problem, we proposed a new approach for malware detection in [23,21] that consists of (1) Modeling the program using a Pushdown System (PDS). This

* Work partially funded by ANR grant ANR-08-SEGI-006.

allows us to track the behavior of the stack. (2) Introducing a new logic, called SCTPL, to specify malicious behaviors. SCTPL can be seen as an extension of the branching-time temporal logic CTL with variables, quantifiers, and regular predicates over the stack. Extension with variables and quantifiers allows to express malicious behaviors in a more succinct way and regular predicates allow to specify properties on the stack content which is important for malware detection. (3) And reducing the malware detection problem to the model-checking problem of PDSs against SCTPL formulas. Our techniques were implemented in a tool and applied to detect several viruses.

However, using the techniques of [23,21], the analysis of several malwares runs out of memory due to the complexity of SCTPL model-checking for PDSs. By looking carefully at the SCTPL formulas specifying the malicious behaviors, we found that most of these SCTPL formulas can be expressed in a more precise manner using the Linear Temporal Logic (LTL). Since LTL can express some malicious behaviors that cannot be expressed by SCTPL, and since the complexity of LTL model-checking for PDSs is polynomial in the size of PDSs, whereas the complexity of CTL model-checking for PDSs is exponential, we will apply in this work LTL model-checking for malware detection (instead of applying SCTPL model-checking as we did in [23,21], since this technique lacks precision and runs out of memory in several cases.). To obtain succinct LTL formulas that express malicious behaviors, we follow the idea of [23,21] and introduce the SLTPL logic, an extension of LTL with variables, quantifiers and regular predicates over the stack content. SLTPL is as expressive as LTL with regular valuations [10,16], but it allows to express malicious behaviors in a more succinct way. We show that SLTPL model-checking for PDSs is polynomial in the size of PDSs and we reduce the malware detection problem to SLTPL model-checking for PDSs.

We use the approach of [21] to model a program as a PDS, in which the PDS control locations correspond to the program's control points, and the PDS's stack mimics the program's execution stack. This approach allows to track the program's stack.

In SLTPL, propositions can be predicates of the form $p(x_1, \ldots, x_n)$, where the x_i's are free variables or constants. Free variables can get their values from a finite domain. Variables can be universally or existentially quantified. SLTPL without predicates over the stack content (called LTPL) is as expressive as LTL, but it allows to express malicious behaviors in a more succinct way. For example, consider the statement "There is a register assigned by 0, and then, the content of this register is pushed onto the stack." This statement can be expressed in LTL as a large formula enumerating all the possible registers as follows:

$$(mov(eax, 0) \wedge \mathbf{X}\ push(eax)) \vee$$
$$(mov(ebx, 0) \wedge \mathbf{X}\ push(ebx)) \vee$$
$$(mov(ecx, 0) \wedge \mathbf{X}\ push(ecx)) \vee \ldots$$

where every instruction is regarded as a predicate, e.g., $mov(eax, 0)$ is a predicate. However, this LTL formula is large for such a simple statement. Using LTPL, this can be expressed by $\exists r\ (mov(r, 0) \wedge \mathbf{X}\ push(r))$ which expresses in a succinct way that there exists a *register* r s.t. the above holds.

However, LTPL cannot specify properties about the stack, which is important for malware detection as explained previously. For example, consider Figure 1(a). It corresponds to a critical fragment of the Trojan LdPinch [13] that adds itself into the registry key listing to get started at boot time. To do this, it calls the API function *GetModuleFileNameA* with 0 and an address a as parameters [1]. After calling this function, the

[1] Parameters to a function in assembly are passed by pushing them onto the stack before a call to the

file name of its own executable will be stored in the address a. Then, the API function *RegSetValueExA* is called with a as parameter (i.e., its own file name). This adds its file name into the registry key listing. We cannot specify this malicious behavior in a precise manner using LTPL. Indeed, a virus writer can easily use some obfuscation techniques in order to escape from any LTPL specification. E.g., let us introduce one *push* followed by one *pop* after *push 0* at line l_2 as done in Figure 1(b). This fragment has the same malicious behavior than the fragment in Figure 1(a). Since the number of pushes and pops can be arbitrary, it is always possible for virus writers to change their code in order to escape from a given LTPL formula. To overcome this problem, we introduce SLTPL, which is extension of LTPL with regular predicates over the stack. Such predicates are given by Regular Variable Expressions over the stack alphabet and some free variables (which can also be existentially and universally quantified). SLTPL is as expressive as LTL with regular valuations [10], but more succinct. In this setting, the malicious behavior of Figures 1(a) and (b) can be specified as follows: $\mathbf{F}\ \exists a\Big(call(GetModuleFileNameA) \wedge 0\ a\ \Gamma^* \wedge \mathbf{F}(call(RegSetValueExA) \wedge a\ \Gamma^*)\Big)$, where $0\ a\ \Gamma^*$ (resp. $a\ \Gamma^*$) is a regular predicate expressing that the top of the stack are 0 and a (resp. a). The SLTPL formula states that there exists a path in which *GetModuleFileNameA* is called with 0 and some address a as parameters (i.e., 0 and a are on the top of the stack), later *RegSetValueExA* is called with a as parameter. This specification can detect both fragments in Figure 1(a) and (b), because it allows to specify the content of the stack when *GetModuleFileNameA* is called. Note that it is important to use PDSs as a model in order to have specifications with predicates over the stack.

l_1: push a
l_2: push 0
l_3: call GetModuleFileNameA
l_4: push a
l_5: call RegSetValueExA

(a)

l'_1: push a
l'_2: push 0
l'_3: push eax
l'_4: pop eax
l'_5: call GetModuleFileNameA
l'_6: push a
l'_7: call RegSetValueExA

(b)

Fig. 1. (a) A fragment of the Trojan LdPinch and (b) The obfuscated version.

Thus, we reduce the malware detection problem to the SLTPL model checking problem for PDSs. To solve this problem, we first present a reduction from LTPL model-checking for PDSs to the emptiness problem of Symbolic Büchi PDSs (SBPDS). This latter problem can be efficiently solved by [11]. Then, we consider the SLTPL model checking problem for PDSs. We introduce Extended Finite Automata (EFA) to represent regular predicates. To perform SLTPL model-checking, we first construct a *Symbolic PDS* which is a kind of synchronization of the PDS and the EFAs that allows to determine whether the stack predicates hold at a given step by looking only at the top of the stack of the symbolic PDS. This allows us to reduce the SLTPL model-checking problem for PDSs to the emptiness problem of SBPDSs.

We implemented our techniques in a tool and applied it to detect several malwares. Our tool can detect all the malwares that we considered. The experimental results show that detecting malware using SLTPL model-checking performs better than using SCTPL model-checking [23,21] and LTL model-checking for PDSs with regular valuations [10]. Moreover, the analysis of several examples terminated using SLTPL model-checking, while it runs out of memory/time using SCTPL or LTL with regular valuations model-checking. Moreover, some malicious behaviors as expressed in [23,21] produce some false alarms. Using SLTPL, these false alarms are avoided. Our tool can also detect the notorious malware *Flame* that was undetected for more than five years.

function is made. The code in the called function later retrieves these parameters from the stack.

Related Work: Quantified (Propositional) Linear Temporal Logic (QPLTL, QLTL) [20] is close to LTPL. However, QPLTL does not allow to quantify over parameters of atomic propositions. LTPL is a subclass of the First-order Linear Temporal Logic (FO-LTL) [14]. [14] considers only the satisfiability problem of FO-LTL and its fragments rather than the model-checking problem. $\mathcal{L}_{\mathcal{MDG}}$ [26] and $\mathcal{L}_{\mathcal{MDG}^*}$ [24] are two sub-logics of FO-LTL. However, $\mathcal{L}_{\mathcal{MDG}}$ disallows temporal operator nesting and properties beyond its templates, and $\mathcal{L}_{\mathcal{MDG}^*}$ cannot use existential and universal operators. FO-LTL was used for malware detection in [3]. All these works cannot specify predicates over the stack.

Model-checking and static analysis such as [4,19,7,8,15,1,2] have been applied to detect malicious behaviors. 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 explained in the introduction, being able to track the stack is important for many malicious behaviors. [18] keeps track of the stack by computing an abstract stack graph which finitely represents the infinite set of all the possible stacks for every control point of the program. Their technique can detect some malicious behaviors that change the stack. However, they cannot specify the other malicious behaviors that SLTPL can describe. [17] performs context-sensitive analysis of *call* and *ret* obfuscated binaries. They use abstract interpretation to compute an abstraction of the stack. We believe that our techniques are more precise since we do not abstract the stack. Moreover, the techniques of [17] were only tried on toy examples, they have not been applied for malware detection.

CTPL [15] is an extension of CTL with variables and quantifiers. SCTPL [23,21] is an extension of CTPL with predicates over the stack content. CTL, CTPL and SCTPL are incomparable with LTPL or SLTPL. For malware detection, experimental results show that SLTPL model-checking performs better and is more precise.

Outline. Section 2 gives the definition of PDSs. Section 3 introduces LTPL and SLTPL. LTPL and SLTPL model-checking for PDSs are given in Sections 4 and 5, respectively. Experimental results are shown in Section 6. Due to lack of space, proofs and the full table of experiments are omitted. They can be found in the full version of the paper [22].

2 Binary Code Modeling

In this section, we recall the definition of pushdown systems. We use the translation of [21] to model binary programs as pushdown systems.

A *Pushdown System* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations, Γ is the stack alphabet, and $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules. 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 $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ instead.

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 an immediate successor of c iff $c \rightsquigarrow_{\mathcal{P}} c'$. An execution of $\mathcal{P}$ is a sequence of configurations $\pi = c_0 c_1 ...$ s.t. $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$ for every $i \geq 0$. Let $\pi(i)$ denote c_i and π^i denote the *suffix* of π starting from $\pi(i)$. For technical reasons, w.l.o.g., we assume that for every transition rule $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, $|\omega| \leq 2$ (see [11]).

3 Malicious Behavior Specification

We define the Stack Linear Temporal Predicate Logic (SLTPL) as an extension of the Linear Temporal Logic (LTL) with variables and regular predicates over the stack content.

Variables are parameters of atomic predicates and can be quantified by the existential and universal operators. Regular predicates are represented by regular variable expressions and are used to specify the stack content of the PDS.

3.1 Environments, Predicates and Regular Variable Expressions

From now on, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, ...\}$ be a finite set of variables ranging over a finite domain $\mathcal{D}$. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment that assigns a value $c \in \mathcal{D}$ to each variable $x \in \mathcal{X}$ s.t. $B(c) = c$ for every $c \in \mathcal{D}$. $B[x \leftarrow c]$ denotes the environment s.t. $B[x \leftarrow c](x) = c$ and $B[x \leftarrow c](y) = B(y)$ for every $y \neq x$. Let $\mathcal{B}$ be the set of all the environments. Let $\Theta_{id} = \{(B_1, B_2) \in \mathcal{B} \times \mathcal{B} \mid B_1 = B_2\}$ be the identity relation for environments, and for every $x \in \mathcal{X}$, $\Theta_x = \{(B_1, B_2) \in \mathcal{B} \times \mathcal{B} \mid \forall x' \in \mathcal{X}\ s.t.\ x \neq x', B_1(x') = B_2(x')\}$ be the relation that abstracts away the value of x.

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

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, a finite set $\mathcal{R}$ of *Regular Variable Expressions* (RVEs) e over $\mathcal{X} \cup \Gamma$ is defined by: $e ::= \epsilon \mid a \in \mathcal{X} \cup \Gamma \mid e + e \mid e \cdot e \mid e^*$. The language $L(e)$ of a RVE e is a subset of $P \times \Gamma^* \times \mathcal{B}$ defined inductively as follows: $L(\epsilon) = \{(\langle p, \epsilon\rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(x)$, where $x \in \mathcal{X}$ is the set $\{(\langle p, \gamma\rangle, B) \mid p \in P, \gamma \in \Gamma, B \in \mathcal{B} : B(x) = \gamma\}$; $L(\gamma)$, where $\gamma \in \Gamma$ is the set $\{(\langle p, \gamma\rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(e_1 + e_2) = L(e_1) \cup L(e_2)$; $L(e_1 \cdot e_2) = \{(\langle p, \omega_1\omega_2\rangle, B) \mid (\langle p, \omega_1\rangle, B) \in L(e_1);\ (\langle p, \omega_2\rangle, B) \in L(e_2)\}$; $L(e^*) = \{(\langle p, \omega\rangle, B) \mid \omega \in \{u \in \Gamma^* \mid (\langle p, u\rangle, B) \in L(e)\}^*\}$.

3.2 The Stack Linear Temporal Predicate Logic

A SLTPL formula is a LTL formula where predicates and RVEs are used as atomic propositions, and where quantifiers over variables are used. 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). We use the *release operator* $\mathbf{R}$ as the dual of the until operator $\mathbf{U}$. Formally, the set of SLTPL formulas is given by (where $x \in \mathcal{X}$, $e \in \mathcal{R}$ and $b(\alpha_1, ..., \alpha_m) \in AP_{\mathcal{X}}$):

$$\varphi ::= b(\alpha_1, ..., \alpha_m) \mid \neg b(\alpha_1, ..., \alpha_m) \mid e \mid \neg e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x\, \varphi \mid \exists x\, \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi \mid \varphi \mathbf{R} \varphi$$

The other standard operators of LTL can be expressed by the above operators: $\mathbf{F}\psi = true\mathbf{U}\psi$ and $\mathbf{G}\psi = false\mathbf{R}\psi$. A SLTPL formula ψ is a LTPL formula iff the formula ψ does not use any regular predicate $e \in \mathcal{R}$. A variable x is a *free variable* of ψ if it is out of the scope of a quantification in ψ.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, let $\lambda : AP_{\mathcal{D}} \rightarrow 2^P$ be a labeling function that assigns a set of control locations to each predicate. Let $c = \langle p, \omega\rangle$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SLTPL formula ψ in c (denoted by $c \models_\lambda \psi$) iff there exists an environment $B \in \mathcal{B}$ s.t. c satisfies ψ under B (denoted by $c \models^B_\lambda \psi$). $c \models^B_\lambda \psi$ holds iff there exists an execution π starting from c s.t. π satisfies ψ under B (denoted by $\pi \models^B_\lambda \psi$), where $\pi \models^B_\lambda \psi$ is defined by induction as follows:

- $\pi \models^B_\lambda b(\alpha_1, ..., \alpha_m)$ iff the control location p of $\pi(0)$ is in $\lambda(b(B(\alpha_1), ..., B(\alpha_m)))$;
- $\pi \models^B_\lambda \neg b(\alpha_1, ..., \alpha_m)$ iff the control location p of $\pi(0)$ is not in $\lambda(b(B(\alpha_1), ..., B(\alpha_m)))$;
- $\pi \models^B_\lambda e$ iff $(\pi(0), B) \in L(e)$;
- $\pi \models^B_\lambda \neg e$ iff $(\pi(0), B) \notin L(e)$;
- $\pi \models^B_\lambda \psi_1 \wedge \psi_2$ iff $\pi \models^B_\lambda \psi_1$ and $\pi \models^B_\lambda \psi_2$;
- $\pi \models^B_\lambda \psi_1 \vee \psi_2$ iff $\pi \models^B_\lambda \psi_1$ or $\pi \models^B_\lambda \psi_2$;

- $\pi \models^B_\lambda \forall x\, \psi$ iff for every $v \in \mathcal{D}$, $\pi \models^{B[x\leftarrow v]}_\lambda \psi$;
- $\pi \models^B_\lambda \exists x\, \psi$ iff there exists $v \in \mathcal{D}$ s.t. $\pi \models^{B[x\leftarrow v]}_\lambda \psi$;
- $\pi \models^B_\lambda \mathbf{X}\, \psi$ iff $\pi^1 \models^B_\lambda \psi$;
- $\pi \models^B_\lambda \psi_1 \mathbf{U} \psi_2$ iff there exists $i \geq 0$ s.t. $\pi^i \models^B_\lambda \psi_2$ and $\forall j,\ 0 \leq j < i : \pi^j \models^B_\lambda \psi_1$;
- $\pi \models^B_\lambda \psi_1 \mathbf{R} \psi_2$ iff for all $j \geq 0$, if for any $i < j : \pi^i \not\models^B_\lambda \psi_1$, then $\pi^j \models^B_\lambda \psi_2$.

Given a SLTPL formula ψ, let $cl_\exists(\psi)$ (resp. $cl_\forall(\psi)$ and $cl_{\mathbf{U}}(\psi)$) denote the set of $\exists$-formulas (resp. $\forall$-formulas and $\mathbf{U}$-formulas) of the form $\exists x\phi$ (resp. $\forall x\phi$ and $\phi_1 \mathbf{U} \phi_2$) of ψ. Let $cl(\psi)$ be the *closure* of ψ defined as the smallest set of formulas containing ψ and satisfying the following: if $\phi_1 \wedge \phi_2 \in cl(\psi)$ or $\phi_1 \vee \phi_2 \in cl(\psi)$, then $\phi_1 \in cl(\psi)$ and $\phi_2 \in cl(\psi)$; if $\mathbf{X}\phi_1 \in cl(\psi)$, or $\exists x\phi_1 \in cl(\psi)$, or $\forall x\phi_1 \in cl(\psi)$ or $\neg\phi_1 \in cl(\psi)$, then $\phi_1 \in cl(\psi)$; if $\phi_1 \mathbf{U} \phi_2 \in cl(\psi)$, then $\phi_1 \in cl(\psi), \phi_2 \in cl(\psi)$ and $\mathbf{X}(\phi_1 \mathbf{U} \phi_2) \in cl(\psi)$; if $\phi_1 \mathbf{R} \phi_2 \in cl(\psi)$, then $\phi_1 \in cl(\psi), \phi_2 \in cl(\psi)$ and $\mathbf{X}(\phi_1 \mathbf{R} \phi_2) \in cl(\psi)$.

LTL with regular valuations is an extension of LTL where the atomic propositions can be regular sets of configurations over the stack alphabet [10,16]. SLTPL is as expressive as LTL with regular valuations. Moreover, since the domain $\mathcal{D}$ is finite, we have:

Proposition 1. *LTPL and LTL (resp. SLTPL and LTL with regular valuations) have the same expressive power. SLTPL is more expressive than LTL.*

3.3 Modeling Malicious Behaviors Using SLTPL

Due to lack of space, we show only one typical malicious behavior here: windows viruses that compute the entry address of Kernel32.dll. We show that this behavior can be expressed in a more precise manner using SLTPL instead of SCTPL, and that if we use SCTPL to describe it, we can obtain false alarms than can be avoided when using SLTPL (see Table 1). More malicious behaviors can be found in the full version of the paper [22].

Kernel32.dll base address viruses: Many Windows viruses use API functions 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 the Kernel32.dll export section and find the entry addresses of the API functions. For this, the virus looks first for the DOS header (the first word of the DOS header is *5A4Dh* in hex (*MZ* in ascii)); and then looks for the PE header (the first two words of the PE header is *4550h* in hex (*PE*00 in ascii)). Figure 2(a) presents a disassembled code fragment performing this malicious behavior. This behavior can be specified in SLTPL using the formula $\Psi_{wv} = \mathbf{GF}(\exists r_1\ cmp(r_1, 5A4Dh) \wedge \mathbf{F}\ \exists r_2\ cmp(r_2, 4550h))$. This SLTPL formula expresses that the program has a loop such that there are two variables r_1 and r_2 such that first, r_1 is compared to *5A4Dh*, and then r_2 is compared to *4550h*. This formula can detect the malware in Figure 2(a). It can be shown that there is no CTL-like formula equivalent to Ψ_{wv} [9]. In [23,21], to be able to express this malicious behavior using a CTL-like formula, we used the following formula: $\Psi'_{wv} = \mathbf{EGEF}(\exists r_1\ cmp(r_1, 5A4Dh) \wedge \mathbf{EF}\ \exists r_2\ cmp(r_2, 4550h))$. This formula can detect the malware in Figure 2(a). However, the benign program in Figure 2(b) that compares with *5A4Dh*

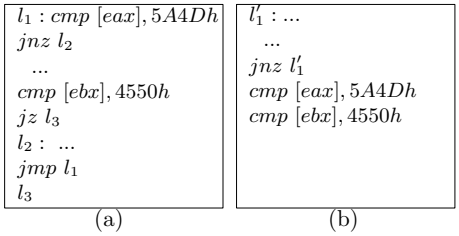

Fig. 2.

and *4550h* only *once* is also detected as a malware using Ψ'_{wv} due to the loop at l'_1, while Ψ_{wv} will not detect it as a malware. In our experiments, as shown in Table 1, several benign programs are detected as malwares using Ψ'_{wv}, whereas Ψ_{wv} classified them as benign programs.

4 LTPL Model-Checking for PDSs

In this section, we show how to reduce LTPL model-checking for PDSs to the emptiness problem of symbolic Büchi PDSs which can be efficiently solved by [11].

4.1 Symbolic Büchi Pushdown Systems

A *Symbolic Pushdown System* (SPDS) $\mathcal{P}$ is a tuple (P, Γ, Δ), where P is a finite set of control locations, Γ is the stack alphabet and Δ is a set of symbolic transition rules of the form $\langle p, \gamma\rangle \overset{\Theta}{\hookrightarrow} \langle q, \omega\rangle$ s.t. $p, q \in P, \gamma \in \Gamma, \omega \in \Gamma^*$, and $\Theta \subseteq \mathcal{B} \times \mathcal{B}$.

A *symbolic* transition $\langle p, \gamma\rangle \overset{\Theta}{\hookrightarrow} \langle q, \omega\rangle$ denotes the following set of PDS transition rules: $\langle (p, B), \gamma\rangle \hookrightarrow \langle (q, B'), \omega\rangle$ for every $B, B' \in \mathcal{B}$ s.t. $(B, B') \in \Theta$. For every $\omega' \in \Gamma^*$, $\langle (q, B'), \omega\omega'\rangle$ is an immediate successor of $\langle (p, B), \gamma\omega'\rangle$, denoted by $\langle (p, B), \gamma\omega'\rangle \rightsquigarrow_{\mathcal{P}} \langle (q, B'), \omega\omega'\rangle$. A run (execution) of $\mathcal{P}$ from $\langle (p_0, B_0), \omega_0\rangle$ is a sequence $\langle (p_0, B_0), \omega_0\rangle\langle (p_1, B_1), \omega_1\rangle \cdots$ over $P \times \mathcal{B} \times \Gamma^*$ s.t. for every $i \geq 0$, $\langle (p_i, B_i), \omega_i\rangle \rightsquigarrow_{\mathcal{P}} \langle (p_{i+1}, B_{i+1}), \omega_{i+1}\rangle$.

A *Symbolic Büchi Pushdown System* (SBPDS) $\mathcal{BP}$ is a tuple (P, Γ, Δ, F), where (P, Γ, Δ) is a SPDS and $F \subseteq P$ is a finite set of accepting control locations. A run of the SBPDS $\mathcal{BP}$ is accepting iff it infinitely often visits some control locations in F. Let $L(\mathcal{BP})$ be the set of configurations $\langle (p, B), \omega\rangle \in P \times \mathcal{B} \times \Gamma^*$ from which $\mathcal{BP}$ has an accepting run.

Theorem 1. *Given a SBPDS $\mathcal{BP} = (P, \Gamma, \Delta, F)$, for every configuration $\langle (p, B), \omega\rangle \in P \times \mathcal{B} \times \Gamma^*$, whether or not $\langle (p, B), \omega\rangle$ is in $L(\mathcal{BP})$ can be decided in time* $O(|P| \cdot |\Delta|^2 \cdot |\mathcal{D}|^{3|\mathcal{X}|})$.

Given a SBPDS $\mathcal{BP}$ with n control locations, m boolean variables and d transition rules, [11] shows that $L(\mathcal{BP})$ can be computed in time $O(n \cdot 2^{3m} \cdot d^2)$. We can use $|\mathcal{X}| \cdot \log_2 |\mathcal{D}|$ boolean variables to represent the set of variables $\mathcal{X}$ over $\mathcal{D}$. Thus, we can decide whether $\langle (p, B), \omega\rangle$ is in $L(\mathcal{BP})$ in time $O(|P| \cdot |\Delta|^2 \cdot |\mathcal{D}|^{3|\mathcal{X}|})$.

A *Generalized Symbolic Büchi PDS* (GSBPDS) $\mathcal{BP}$ is a tuple (P, Γ, Δ, F), where (P, Γ, Δ) is a SPDS and $F = \{F_1, ..., F_k\}$ is a set of sets of accepting control locations. A run of the GSBPDS $\mathcal{BP}$ is accepting iff for every i, $1 \leq i \leq k$, the run infinitely often visits some control locations in F_i. Let $L(\mathcal{BP})$ denote the set of configurations $\langle (p, B), \omega\rangle \in P \times \mathcal{B} \times \Gamma^*$ from which the GSBPDS $\mathcal{BP}$ has an accepting run.

Proposition 2. *Given a GSBPDS $\mathcal{BP}$, we can obtain a SBPDS $\mathcal{BP}'$ s.t. $L(\mathcal{BP}) = L(\mathcal{BP}')$.*

The translation from GSBPDS to SBPDS is similar to the translation from generalized Büchi finite automata into Büchi finite automata (see [9]). The translation expands the size by a factor of $k + 1$, where k denotes the number of sets in F.

4.2 From LTPL Model-Checking for PDSs to the Emptiness Problem of SBPDSs

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, $\lambda : AP_{\mathcal{D}} \to 2^P$ a labeling function, ψ a LTPL formula. We construct a GSBPDS $\mathcal{BP}_\psi$ s.t. $\mathcal{BP}_\psi$ has an accepting run from $\langle (\langle\!|p, \{\psi\}|\!\rangle, B), \omega\rangle$ iff $\mathcal{P}$ has an execution π from $\langle p, \omega\rangle$ s.t. π satisfies ψ under B. Thus, $\langle p, \omega\rangle \models_\lambda \psi$ iff there exists $B \in \mathcal{B}$ s.t. $\mathcal{BP}_\psi$ has an accepting run from $\langle (\langle\!|p, \{\psi\}|\!\rangle, B), \omega\rangle$ (since $\langle p, \omega\rangle \models_\lambda \psi$ iff there exists $B \in \mathcal{B}$ s.t. $\langle p, \omega\rangle \models^B_\lambda \psi$). Let $cl_{\mathbf{U}}(\psi) = \{\phi_1 \mathbf{U} \varphi_1, ..., \phi_k \mathbf{U} \varphi_k\}$ be the set of $\mathbf{U}$-formulas of $cl(\psi)$. We define $\mathcal{BP}_\psi = (P', \Gamma, \Delta', F)$ as follows: $P' = P \times 2^{cl(\psi)}$, $F = \{P \times F_{\phi_1 \mathbf{U} \varphi_1}, ..., P \times F_{\phi_k \mathbf{U} \varphi_k}\}$, where for every i, $1 \leq i \leq k$, $F_{\phi_i \mathbf{U} \varphi_i} = \{\Phi \subseteq cl(\psi) \mid \text{if } \phi_i \mathbf{U} \varphi_i \in \Phi \text{ then } \varphi_i \in \Phi\}$, and Δ' is the smallest set of transition rules satisfying the following: for every $\Phi \subseteq cl(\psi)$, $p \in P, \gamma \in \Gamma$,

(α_1): if $\phi = b(x_1, ..., x_m) \in \Phi$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p, \Phi \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$, where $\Theta = \{(B, B) \mid B \in \mathcal{B} \wedge p \in \lambda(b(B(x_1), ..., B(x_m)))\}$;

(α_2): if $\phi = \neg b(x_1, ..., x_m) \in \Phi$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p, \Phi \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$, where $\Theta = \{(B, B) \mid B \in \mathcal{B} \wedge p \notin \lambda(b(B(x_1), ..., B(x_m)))\}$;

(α_3): if $\phi = \phi_1 \wedge \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;

(α_4): if $\phi = \phi_1 \vee \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ and $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;

(α_5): if $\phi = \exists x \phi_1 \in \Phi$, then:

- ($\alpha_{5.1}$): if x is not a free variable of any formula in Φ, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_x}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;
- ($\alpha_{5.2}$): otherwise, for every $c \in \mathcal{D}$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_c\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$, where ϕ_c is ϕ_1 where x is substituted by c;

(α_6): if $\phi = \forall x \phi_1 \in \Phi$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_c \mid c \in \mathcal{D}\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$, where ϕ_c is ϕ_1 where x is substituted by c;

(α_7): if $\phi = \phi_1 \mathbf{U} \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ and $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1, \mathbf{X}\phi\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;

(α_8): if $\phi = \phi_1 \mathbf{R} \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ and $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_2, \mathbf{X}\phi\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$;

(α_9): if $\Phi = \{\mathbf{X}\phi_1, ..., \mathbf{X}\phi_m\}$, for every $\langle p, \gamma\rangle \hookrightarrow \langle p', \omega\rangle \in \Delta$, $\langle (\!|p, \Phi|\!), \gamma\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p', \{\phi_1, ..., \phi_m\}|\!), \omega\rangle \in \Delta'$.

Intuitively, $\mathcal{BP}_\psi$ is a kind of "product" of $\mathcal{P}$ and ψ. $\mathcal{BP}_\psi$ has an accepting run from $\langle (\!|p, \{\psi\}|\!), B), \omega\rangle$ iff $\mathcal{P}$ has an execution π starting from $\langle p, \omega\rangle$ s.t. π satisfies ψ under B. The control locations of $\mathcal{BP}_\psi$ are of the form $(\!|p, \Phi|\!)$, where Φ is a set of formulas, because the satisfiability of a single formula ϕ may depend on several other formulas. E.g., the satisfiability of $\phi_1 \wedge \phi_2$ depends on ϕ_1 and ϕ_2. Thus, we have to store a set of formulas into the control locations of $\mathcal{BP}_\psi$. The intuition behind each rule is explained as follows. (By abuse of notation, given a set of formulas Φ, we write $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda \phi$ for every $\phi \in \Phi$.) Let π be an execution of $\mathcal{P}$ from $\langle p, \omega\rangle$.

If $b(x_1, ..., x_m) \in \Phi$, then $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda b(x_1, ..., x_m)$ (i.e., $p \in \lambda(b(B(x_1), ..., B(x_m)))$) and π satisfies all the other formulas of Φ under B (i.e., $\pi \models^B_\lambda \Phi \setminus \{b(x_1, ..., x_m)\}$). This is ensured by Item (α_1) stating that $\mathcal{BP}_\psi$ has an accepting run from $\langle (\!|p, \Phi|\!), B), \omega\rangle$ iff $p \in \lambda(b(B(x_1), ..., B(x_m)))$ and $\mathcal{BP}_\psi$ has an accepting run from $\langle (\!|p, \Phi \setminus \{b(x_1, ..., x_m)\}|\!), B), \omega\rangle$. Item ($\alpha_2$) is similar to Item ($\alpha_1$).

If $\phi_1 \wedge \phi_2 \in \Phi$, then, $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda \phi_1$, $\pi \models^B_\lambda \phi_2$ and π satisfies all the other formulas of Φ under B (i.e., $\pi \models^B_\lambda \Phi \setminus \{\phi_1 \wedge \phi_2\}$). This is ensured by Item (α_3) expressing that $\mathcal{BP}_\psi$ has an accepting run from $\langle (\!|p, \Phi|\!), B), \omega\rangle$ iff $\mathcal{BP}_\psi$ has an accepting run from $\langle (\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi_1 \wedge \phi_2\}|\!), B), \omega\rangle$. Item ($\alpha_4$) is analogous.

If $\phi_1 \mathbf{U} \phi_2 \in \Phi$, then, $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda \phi_2$ holds or both ($\pi \models^B_\lambda \phi_1$ and $\pi \models^B_\lambda \mathbf{X}(\phi_1 \mathbf{U} \phi_2)$) hold, and π satisfies all the other formulas of Φ under B (i.e., $\pi \models^B_\lambda \Phi \setminus \{\phi_1 \mathbf{U} \phi_2\}$). This is ensured by Item (α_7). Since ϕ_2 should eventually hold, to prevent the case where the run of $\mathcal{BP}_\psi$ always carries ϕ_1 and $\mathbf{X}(\phi_1 \mathbf{U} \phi_2)$ and never ϕ_2, we set $P \times F_{\phi_1 \mathbf{U} \phi_2} = P \times \{\Phi' \subseteq cl(\psi) \mid$ if $\phi_1 \mathbf{U} \phi_2 \in \Phi'$ then $\phi_2 \in \Phi'\}$ as a set of accepting control locations. Then, the accepting run of $\mathcal{BP}_\psi$ will infinitely often visit some control locations in $P \times F_{\phi_1 \mathbf{U} \phi_2}$ which guarantees that ϕ_2 eventually holds. Item (α_8) is similar to Item (α_7).

If $\Phi = \{\mathbf{X}\phi_1, ..., \mathbf{X}\phi_m\}$, then $\pi \models^B_\lambda \Phi$ iff $\pi^1 \models^B_\lambda \{\phi_1, ..., \phi_m\}$. This is ensured by Item (α_9) which expresses that $\mathcal{BP}_\psi$ has an accepting run from $\langle (\!|p, \Phi|\!), B), \omega\rangle$ iff $\mathcal{BP}_\psi$ has an accepting run from $\langle (\!|p', \{\phi_1, ..., \phi_m\}|\!), B), \omega'\rangle$ where $\langle p', \omega'\rangle = \pi(1)$, i.e., $\langle p', \omega'\rangle$ is the immediate successor of $\langle p, \omega\rangle$ in π. Note that if there is a formula $\phi \in \Phi$ that is not in

the form of $\mathbf{X}\phi'$, we have to handle ϕ first by Items $(\alpha_1), ..., (\alpha_8)$, as ϕ depends on π not π^1. This is why we do not consider the case $\mathbf{X}\phi \in \Phi$, and consider only the case where $\Phi = \{\mathbf{X}\phi_1, ..., \mathbf{X}\phi_m\}$.

If $\forall x\phi \in \Phi$, then $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda \forall x\phi$ and $\pi \models^B_\lambda \Phi \setminus \{\forall x\phi\}$. Since $\pi \models^B_\lambda \forall x\phi$ iff $\pi \models^B_\lambda \bigwedge_{c\in\mathcal{D}} \phi_c$, where ϕ_c is ϕ_1 where x is substituted by c, we replace $\forall x\phi$ by $\bigwedge_{c\in\mathcal{D}} \phi_c$. This is expressed by Item (α_6) stating that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi|\!), B), \omega\rangle$ iff $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi \cup \{\phi_c \mid c \in \mathcal{D}\} \setminus \{\forall x\phi\}|\!), B), \omega\rangle$.

If $\exists x\phi \in \Phi$, then the construction depends on whether x is a free variable of some formula in Φ or not:

- if x is not a free variable of any formula in Φ, then $\pi \models^B_\lambda \Phi$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models^{B[x\leftarrow c]}_\lambda \phi$ (i.e., $\pi \models^B_\lambda \exists x\phi$) and $\pi \models^B_\lambda \Phi \setminus \{\exists x\phi\}$. Since x is not a free variable of any formula in Φ, we can get that $\pi \models^B_\lambda \Phi \setminus \{\exists x\phi\}$ iff $\pi \models^{B[x\leftarrow c]}_\lambda \Phi \setminus \{\exists x\phi\}$ for every $c \in \mathcal{D}$. This implies that $\pi \models^B_\lambda \Phi$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models^{B[x\leftarrow c]}_\lambda \phi$ and $\pi \models^{B[x\leftarrow c]}_\lambda \Phi \setminus \{\exists x\phi\}$. This is ensured by Item $(\alpha_{5.1})$ stating that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi|\!), B), \omega\rangle$ iff there exists $c \in \mathcal{D}$ s.t. $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi \cup \{\phi\} \setminus \{\exists x\phi\}|\!), B[x \leftarrow c]), \omega\rangle$, since $(B, B[x \leftarrow c]) \in \Theta_x$.
- otherwise, if x is a free variable of some formula φ in Φ, we cannot apply Item $(\alpha_{5.1})$. Indeed, it may happen that ϕ is satisfied only when $x = c$ (i.e., $\pi \models^{B[x\leftarrow c]}_\lambda \phi$), φ is not satisfied when $x = c$ (i.e., $\pi \not\models^{B[x\leftarrow c]}_\lambda \varphi$), whereas $\pi \models^B_\lambda \{\varphi, \exists x\phi\}$. In this case, we apply Item $(\alpha_{5.2})$. Since $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda \bigvee_{c\in\mathcal{D}} \phi_c$ (i.e., $\pi \models^B_\lambda \exists x\phi$) and $\pi \models^B_\lambda \Phi \setminus \{\exists x\phi\}$, where ϕ_c is ϕ where x is substituted by c. Since $\pi \models^B_\lambda \bigvee_{c\in\mathcal{D}} \phi_c$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models^B_\lambda \phi_c$, then, $\pi \models^B_\lambda \Phi$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models^B_\lambda \phi_c$ and $\pi \models^B_\lambda \Phi \setminus \{\exists x\phi\}$. This is ensured by Item $(\alpha_{5.2})$ stating that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi|\!), B), \omega\rangle$ iff there exists $c \in \mathcal{D}$ s.t. $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi \cup \{\phi_c\} \setminus \{\exists x\phi\}|\!), B), \omega\rangle$, as $(B, B) \in \Theta_{id}$. Note that we can use Item $(\alpha_{5.2})$ even in the previous case when x is not a free variable of any formula in Φ. However, it is more efficient to use Item $(\alpha_{5.1})$ in this case, since Item $(\alpha_{5.1})$ adds only one symbolic transition rule, whereas Item $(\alpha_{5.2})$ adds $|\mathcal{D}|$ symbolic transition rules.

Thus, we can show that:

Theorem 2. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : \mathrm{AP}_\mathcal{D} \to 2^P$, and a LTPL formula ψ, we can construct a GSBPDS $\mathcal{BP}_\psi$ with $\mathrm{O}((|\Delta| + |P| \cdot |\Gamma|) \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$ transition rules and $\mathrm{O}(|P| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$ states s.t. for every $B \in \mathcal{B}$ and every configuration $\langle p, \omega\rangle \in P \times \Gamma^*$, $\langle p, \omega\rangle$ satisfies ψ under B iff $\langle((\!|p, \{\psi\}|\!), B), \omega\rangle \in L(\mathcal{BP}_\psi)$.*

Note that we do not need to consider all the possible subsets of $cl(\psi)$ during the construction of $\mathcal{BP}_\psi$. In order to get the above complexity, we can maintain a set of sets of formulas which are reachable from the configuration carrying the set $\{\psi\}$.

From Proposition 2, Theorem 2 and Theorem 1, we get that:

Theorem 3. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : \mathrm{AP}_\mathcal{D} \to 2^P$ and a LTPL formula ψ, for every $B \in \mathcal{B}$ and configuration $\langle p, \omega\rangle$, whether $\langle p, \omega\rangle$ satisfies ψ under B or not can be decided in time $\mathrm{O}(|cl_\mathbf{U}(\psi)| \cdot |P| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot (|\Delta| + |P| \cdot |\Gamma|)^2 \cdot 2^{3|\psi|} \cdot |\mathcal{D}|^{3|\mathcal{X}|})$.*

The complexity follows from the fact that the number of transition rules (resp. states) of the SBPDS equivalent to $\mathcal{BP}_\psi$ is at most $O(|cl_\mathbf{U}(\psi)| \cdot (|\Delta| + |P| \cdot |\Gamma|) \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$ (resp. $O(|cl_\mathbf{U}(\psi)| \cdot |P| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$), and the environments B only need to consider the variables that are used in ψ.

Remark 1. To do LTPL model-checking for PDSs, by Proposition 1, we can translate LTPL formulas into LTL formulas and apply LTL model-checking for PDSs [6,11]. This can be done in time $O(2^{3|\psi|\cdot|\mathcal{D}|^{(|c|\forall(\psi)|+|c|\exists(\psi)|)}})$. Our approach has a better complexity.

5 SLTPL Model-Checking for PDSs

In this section, we show how to do SLTPL model-checking for PDSs. We follow the approach of [10]. Fix a PDS $\mathcal{P}$, a set of variables $\mathcal{X}$ over $\mathcal{D}$ and a SLTPL formula ψ. Roughly speaking, for each RVE e of ψ, we construct a kind of finite automaton $\mathcal{V}$ recognizing all the configurations $(\langle p, \omega\rangle, B) \in P \times \Gamma^* \times \mathcal{B}$ s.t. $(\langle p, \omega\rangle, B) \in L(e)$. Then, we compute a SPDS $\mathcal{P}'$ which is a kind of synchronization of $\mathcal{P}$ and the $\mathcal{V}s$ that allows to determine whether the stack predicates hold at a given step by looking only at the top of the stack of $\mathcal{P}'$. Having $\mathcal{P}'$ allows to readapt the construction of Section 4 and reduce the SLTPL model-checking problem for PDSs to the emptiness problem of SBPDSs.

5.1 Extended Finite Automata

To represent RVEs, we introduce extended finite automata, in which transition rules can be labeled by a set of variables and/or their negations. Formally, let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS and $\xi = \{\alpha, \neg\alpha \mid \alpha \in \Gamma \cup \mathcal{X}\}$, an *Extended Finite Automaton* (EFA) $\mathcal{V}$ is a tuple $(\mathcal{S}, \Lambda, \Gamma, s_0, S_f)$ where $\mathcal{S}$ is a finite set of states, Γ is the input alphabet, $s_0 \in \mathcal{S}$ is the initial state, $S_f \subseteq \mathcal{S}$ is a finite set of final states, and Λ is a finite set of transition rules of the form $s_1 \overset{\ell}{\mapsto} s_2$ s.t. $s_1, s_2 \in \mathcal{S}$, $\ell \subseteq \xi$. Let $B \in \mathcal{B}$ be an environment, $\gamma \in \Gamma$ the input letter, suppose $\mathcal{V}$ is at state s_1 and $t = s_1 \overset{\ell}{\mapsto} s_2$ is a transition rule in Λ, then $\mathcal{V}$ can move to the state s_2 (i.e., s_2 is an immediate successor of s_1 under B over γ), denoted by $s_1 \overset{\gamma}{\twoheadrightarrow}_B s_2$, iff the following conditions hold: (1) for every $\alpha \in \ell$, $B(\alpha) = \gamma$; (2) for every $\neg\alpha \in \ell$, $B(\alpha) \neq \gamma$ (note that $B(\gamma) = \gamma$ if $\gamma \in \Gamma$). Obviously, the transition t will never be fired when either $\gamma_1, \gamma_2 \in \ell \cap \Gamma$ s.t. $\gamma_1 \neq \gamma_2$ or $\alpha, \neg\alpha \in \ell$ for some $\alpha \in \Gamma \cup \mathcal{X}$. This implies that ℓ can contain only one letter from Γ, and for each $\alpha \in \mathcal{X} \cup \Gamma$, ℓ cannot contain both α and $\neg\alpha$. $\mathcal{V}$ recognizes (accepts) a word $\gamma_0 ... \gamma_n$ over Γ under B iff $\mathcal{V}$ has a run $s_0 \overset{\gamma_0}{\twoheadrightarrow}_B s_1 ... s_n \overset{\gamma_n}{\twoheadrightarrow}_B s_{n+1}$ s.t. $s_{n+1} \in S_f$. Let $L(\mathcal{V})$ be the set of all the configurations $(\langle p, \omega\rangle, B) \in P \times \Gamma^* \times \mathcal{B}$ s.t. $\mathcal{V}$ recognizes ω under B.

A EFA $\mathcal{V}$ is *deterministic* (resp. *total*) iff for every state $s \in \mathcal{S}$, environment $B \in \mathcal{B}$, letter $\gamma \in \Gamma$, s has *at most* (resp. *at least*) one immediate successor $s' \in \mathcal{S}$ under B over γ. We can show that:

Proposition 3. *For every EFA $\mathcal{V} = (\mathcal{S}, \Lambda, \Gamma, s_0, S_f)$, we can compute in time $O(2^{|\Lambda|})$ a deterministic and total EFA $\mathcal{V}'$ s.t. $L(\mathcal{V}) = L(\mathcal{V}')$.*

Theorem 4. *For every regular predicate $e \in \mathcal{R}$, we can compute in polynomial time an EFA $\mathcal{V}_e$ s.t. $L(e) = L(\mathcal{V}_e)$.*

Given a configuration $(\langle p, \gamma_1 ... \gamma_m\rangle, B) \in P \times \Gamma^* \times \mathcal{B}$, its *reverse* $(\langle p, \gamma_1 ... \gamma_m\rangle, B)^r$ is the configuration $(\langle p, \gamma_m ... \gamma_1\rangle, B)$. Given a set $L \subseteq P \times \Gamma^* \times \mathcal{B}$, its reverse L^r is the set $\{(\langle p, \gamma_m ... \gamma_1\rangle, B) \mid (\langle p, \gamma_1 ... \gamma_m\rangle, B) \in L\}$. We can show that:

Proposition 4. *For every EFA $\mathcal{V}$, we can get an EFA $\mathcal{V}^r$ in linear time s.t. $L(\mathcal{V})^r = L(\mathcal{V}^r)$.*

Remark 2. To represent RVEs, [23] uses automata with alternating transition rules, called Variable Automata (VA). If we use VAs to represent variable expressions, we will obtain an alternating SBPDS when synchronizing the SLTPL formula with the PDS. We introduce EFAs to avoid using alternation, since checking the emptiness of alternating SBPDSs is exponential in the size of the PDSs [23]. [12] introduces another kind of VAs, which is not suitable for our purpose, since determinizing a VA as defined in [12] is undecidable, but, we need the automata to be deterministic as will be explained later.

5.2 Storing States into the Stack

We fix a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a SLTPL formula ψ. Let $\{e_1, ..., e_n\}$ be the set of RVEs used in ψ. We suppose w.l.o.g. that $\mathcal{P}$ has a bottom-of-stack $\perp \in \Gamma$ that is never popped from the stack. For every $i,\ 1 \leq i \leq n$, let $\mathcal{V}^i = (\mathcal{S}^i, \Lambda^i, \Gamma, s_0^i, S_f^i)$ be a deterministic and total EFA s.t. $L(e_i)^r = L(\mathcal{V}^i)$. Since we have predicates over the stack, to check whether the formula ψ is satisfied, we need to know at each step which RVEs are satisfied by the stack. To this aim, we will compute a SPDS $\mathcal{P}'$ which is a kind of product of $\mathcal{P}$ and the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$, where the states of the $\mathcal{V}^i s$ are stored in the stack of $\mathcal{P}'$. Roughly speaking, the stack alphabet of $\mathcal{P}'$ is of the form $(\gamma, \overrightarrow{S})$, where $\overrightarrow{S} = \left[s_1, \cdots, s_n\right]$, $s_i \in \mathcal{S}^i$ for every $i,\ 1 \leq i \leq n$, is a vector of states of the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$. For every $i,\ 1 \leq i \leq n$, let $\overrightarrow{S}(i)$ denote the i^{th} component of $\overrightarrow{S}$. A configuration $\langle (p, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ is *consistent* iff for every $i,\ 1 \leq i \leq n$, $\mathcal{V}^i$ has a run $\overrightarrow{S_0}(i) \overset{\gamma_0}{\twoheadrightarrow_B} \overrightarrow{S_1}(i) \cdots \overrightarrow{S_{m-1}}(i) \overset{\gamma_{m-1}}{\twoheadrightarrow_B} \overrightarrow{S_m}(i)$ over $\gamma_0 \cdots \gamma_{m-1}$, i.e., the reverse of the stack content $\gamma_{m-1} \cdots \gamma_0$. Intuitively, a consistent configuration $\langle (p, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ denotes that the stack content is $\gamma_m \cdots \gamma_0$ and the runs of the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$ over $\gamma_0 \cdots \gamma_{m-1}$ reach the states $\overrightarrow{S_m}(1), ..., \overrightarrow{S_m}(n)$, respectively (note that $\gamma_0 \cdots \gamma_{m-1}$ is the reverse of the stack content $\gamma_{m-1} \cdots \gamma_0$, this is why the $\mathcal{V}^i s$ are s.t. $L(e_i)^r = L(\mathcal{V}^i)$). For every $i,\ 1 \leq i \leq n$, a consistent configuration $\langle (p, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ satisfies e_i under the environment B iff there exists $s \in S_f^i$ s.t. $\overrightarrow{S_m}(i) \overset{\gamma_m}{\twoheadrightarrow_B} s$. I.e., whether $\langle (p, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ satisfies e_i under B or not depends only on the top of the stack $(\gamma_m, \overrightarrow{S_m})$.

Formally, let $\overrightarrow{\mathcal{S}} = \mathcal{S}^1 \times \cdots \times \mathcal{S}^n$ and $\overrightarrow{S_0} = \left[s_0^1, \cdots, s_0^n\right]$. We compute the SPDS $\mathcal{P}' = (P, \Gamma', \Delta')$ as follows: $\Gamma' = \Gamma \times \overrightarrow{\mathcal{S}}$ is the stack alphabet and the set Δ' of transition rules are defined as follows:

1. $\langle p_1, (\gamma, \overrightarrow{S}) \rangle \overset{\Theta_{id}}{\hookrightarrow} \langle p_2, \epsilon \rangle \in \Delta'$ iff $\langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \epsilon \rangle \in \Delta$ and $\overrightarrow{S} \in \overrightarrow{\mathcal{S}}$;
2. $\langle p_1, (\gamma, \overrightarrow{S}) \rangle \overset{\Theta_{id}}{\hookrightarrow} \langle p_2, (\gamma_1, \overrightarrow{S}) \rangle \in \Delta'$ iff $\langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \gamma_1 \rangle \in \Delta$ and $\overrightarrow{S} \in \overrightarrow{\mathcal{S}}$;
3. $\langle p_1, (\gamma, \overrightarrow{S}) \rangle \overset{\Theta}{\hookrightarrow} \langle p_2, (\gamma_2, \overrightarrow{S'})(\gamma_1, \overrightarrow{S}) \rangle \in \Delta'$ iff $\langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \gamma_2 \gamma_1 \rangle \in \Delta$ and for every $i,\ 1 \leq i \leq n$, $\overrightarrow{S}(i) \overset{\ell_i}{\mapsto} \overrightarrow{S'}(i) \in \Lambda^i$, where $\Theta = \{(B, B) \mid B \in \mathcal{B}, \forall i : 1 \leq i \leq n, (x \in \ell_i \implies B(x) = \gamma_1) \wedge (\neg y \in \ell_i \implies B(y) \neq \gamma_1)\}$.

Intuitively, the run of $\mathcal{P}$ reaches the configuration $\langle p_1, \gamma_m, \cdots \gamma_0 \rangle$ and the runs of the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$ over the stack word $\gamma_0 \cdots \gamma_{m-1}$ reach the states $\overrightarrow{S_m}(1), ..., \overrightarrow{S_m}(n)$, respectively, iff the run of $\mathcal{P}'$ reaches the consistent configuration $\langle (p_1, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$. If $\mathcal{P}$ moves from $\langle p_1, \gamma_m, \cdots \gamma_0 \rangle$ to $\langle p_2, \gamma_{m-1}, \cdots \gamma_0 \rangle$ using the rule $\langle p_1, \gamma_m \rangle \hookrightarrow \langle p_2, \epsilon \rangle$, then the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$ should be at $\overrightarrow{S_{m-1}}(1), ..., \overrightarrow{S_{m-1}}(n)$ after reading the stack word $\gamma_0 \cdots \gamma_{m-2}$, i.e., $\mathcal{P}'$ moves from $\langle (p_1, B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ to $\langle (p_2, B), (\gamma_{m-1}, \overrightarrow{S_{m-1}}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$. This is ensured by Item 1. The intuition behind Item 2 is similar.

If $\mathcal{P}$ moves from $\langle p_1, \gamma'_m \gamma_{m-1} \cdots \gamma_0 \rangle$ to $\langle p_2, \gamma_{m+1} \gamma_m \cdots \gamma_0 \rangle$ using the rule $\langle p_1, \gamma'_m \rangle \hookrightarrow \langle p_2, \gamma_{m+1} \gamma_m \rangle$, then, after reading $\gamma_0 \cdots \gamma_m$, the EFAs $\mathcal{V}^1, ..., \mathcal{V}^n$ should be at $\overrightarrow{S_{m+1}}(1), ..., \overrightarrow{S_{m+1}}(n)$ where for every $i,\ 1 \leq i \leq n$, $\overrightarrow{S_m}(i) \overset{\gamma_m}{\twoheadrightarrow_B} \overrightarrow{S_{m+1}}(i)$. I.e., $\mathcal{P}'$ moves from $\langle (p_1, B), (\gamma'_m, \overrightarrow{S_m})(\gamma_{m-1}, \overrightarrow{S_{m-1}}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$ to $\langle (p_2, B), (\gamma_{m+1}, \overrightarrow{S_{m+1}})(\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0}) \rangle$. This is ensured by Item 3. The relation $\Theta = \{(B, B) \mid B \in \mathcal{B}, \forall i : 1 \leq i \leq n, (x \in \ell_i \implies B(x) = \gamma_m) \wedge (\neg y \in \ell_i \implies B(y) \neq \gamma_m)\}$ in the transition rule

$\langle p_1, (\gamma'_m, \overrightarrow{S_m})\rangle \overset{\Theta}{\hookrightarrow} \langle p_2, (\gamma_{m+1}, \overrightarrow{S_{m+1}})(\gamma_m, \overrightarrow{S_m})\rangle$ guarantees that for every i, $1 \leq i \leq n$, the state $\overrightarrow{S_{m+1}}(i)$ is the immediate successor of the state $\overrightarrow{S_m}(i)$ over γ_m under B in $\mathcal{V}^i$.

The fact that EFAs are deterministic guarantees that the top of the stack can infer the truth of the regular predicates.

5.3 Readapting the Reduction underlying Theorem 2

In this subsection, we show how to reduce the SLTPL model-checking problem for SPDSs to the emptiness problem of SBPDSs by a readaptation of the construction underlying Theorem 2. Let $\mathcal{BP}'_\psi = (P', \Gamma', \Delta'', F)$ be the GSBPDS s.t.: $P' = P \times 2^{cl(\psi)}$, $F = \{P \times F_{\phi_1 \mathbf{U}\varphi_1}, ..., P \times F_{\phi_k \mathbf{U}\varphi_k}\}$, where for every i, $1 \leq i \leq k$, $F_{\phi_i \mathbf{U}\varphi_i} = \{\Phi \subseteq cl(\psi) \mid \phi_i \mathbf{U}\varphi_i \notin \Phi \text{ or } \varphi_i \in \Phi\}$, and Δ'' is the smallest set of transition rules satisfying the following: for every $\Phi \subseteq cl(\psi)$, $p \in P, (\gamma, \overrightarrow{S}) \in \Gamma'$:

(β_1): if $\phi = b(x_1, ..., x_m) \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p, \Phi \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$, where $\Theta = \{(B, B) \mid B \in \mathcal{B} \wedge p \in \lambda(b(B(x_1), ..., B(x_m)))\}$;

(β_2): if $\phi = \neg b(x_1, ..., x_m) \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p, \Phi \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$, where $\Theta = \{(B, B) \mid B \in \mathcal{B} \wedge p \notin \lambda(b(B(x_1), ..., B(x_m)))\}$;

(β_3): if $\phi = \phi_1 \wedge \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$;

(β_4): if $\phi = \phi_1 \vee \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$ and $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_2\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$;

(β_5): if $\phi = \exists x \phi_1 \in \Phi$, then:

- ($\beta_{5.1}$): if x is not a free variable of any formula in Φ, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_x}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta'$;
- ($\beta_{5.2}$): otherwise for every $c \in \mathcal{D}$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_c\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta'$, where ϕ_c is ϕ_1 where x is substituted by c;

(β_6): if $\phi = \forall x \phi_1 \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_c \mid c \in \mathcal{D}\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta'$, where ϕ_c is ϕ_1 where x is substituted by c;

(β_7): if $\phi = \phi_1 \mathbf{U} \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_2\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$ and $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1, \mathbf{X}\phi\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$;

(β_8): if $\phi = \phi_1 \mathbf{R} \phi_2 \in \Phi$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$ and $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta_{id}}{\hookrightarrow} \langle (\!|p, \Phi \cup \{\phi_2, \mathbf{X}\phi\} \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$;

(β_9): if $\Phi = \{\mathbf{X}\phi_1, ..., \mathbf{X}\phi_m\}$, for every $\langle p, (\gamma, \overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle p', \omega\rangle \in \Delta'$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta \cap \Theta_{id}}{\hookrightarrow} \langle (\!|p', \{\phi_1, ..., \phi_m\}|\!), \omega\rangle \in \Delta''$;

(β_{10}): if $\phi = e_i \in \Phi \cap \mathcal{R}$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p, \Phi \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$, where $\Theta = \{(B, B) \mid B \in \mathcal{B}, \exists \overrightarrow{S'}, \overrightarrow{S}(i) \overset{\gamma}{\twoheadrightarrow}_B \overrightarrow{S'}(i) \wedge \overrightarrow{S'}(i) \in S^i_f\}$;

(β_{11}): if $\phi = \neg e_i \in \Phi$ s.t. $e_i \in \mathcal{R}$, $\langle (\!|p, \Phi|\!), (\gamma, \overrightarrow{S})\rangle \overset{\Theta}{\hookrightarrow} \langle (\!|p, \Phi \setminus \{\phi\}|\!), (\gamma, \overrightarrow{S})\rangle \in \Delta''$, where $\Theta = \{(B, B) \mid B \in \mathcal{B}, \exists \overrightarrow{S'}, \overrightarrow{S}(i) \overset{\gamma}{\twoheadrightarrow}_B \overrightarrow{S'}(i) \wedge \overrightarrow{S'}(i) \notin S^i_f\}$.

The intuition behind $\mathcal{BP}'_\psi$ is similar to the one underlying Theorem 2. $\mathcal{P}$ has an execution π starting from $\langle p, \gamma_m, ..., \gamma_0\rangle$ s.t. π satisfies ψ under B iff there exist states $\overrightarrow{S_m}, ..., \overrightarrow{S_0}$ s.t. $\langle ((\!|p, \{\psi\}|\!), B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$ is consistent and $\mathcal{BP}'_\psi$ has an accepting run from $\langle ((\!|p, \{\psi\}|\!), B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$. Items $(\beta_1), ..., (\beta_8)$ are similar to Items $(\alpha_1), ..., (\alpha_8)$. The main differences are Items (β_9), (β_{10}) and (β_{11}).

The relation $\Theta \cap \Theta_{id}$ in Item (β_9) ensures that $\langle((\{\!|p', \{\phi_1, ..., \phi_m\}|\!\}), B), \omega\omega'\rangle$ is an immediate successor of $\langle((\{\!|p, \{\mathbf{X}\phi_1, ..., \mathbf{X}\phi_m\}|\!\}), B), (\gamma, \overrightarrow{S})\omega'\rangle$ in the run of $\mathcal{BP}'_\psi$ iff $\langle(p', B), \omega\omega'\rangle$ is an immediate successor of $\langle(p, B), (\gamma, \overrightarrow{S})\omega'\rangle$ in the corresponding run of $\mathcal{P}'$, as $(B, B) \in \Theta \cap \Theta_{id}$ implies that $(B, B) \in \Theta$. This implies that $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\{\!|p, \{\mathbf{X}\phi_1, ..., \mathbf{X}\phi_m\}|\!\}), B), (\gamma, \overrightarrow{S})\omega'\rangle$ iff $\mathcal{P}'$ has an immediate successor $\langle(p', B), \omega\omega'\rangle$ of $\langle(p, B), (\gamma, \overrightarrow{S})\omega'\rangle$ s.t. $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\{\!|p', \{\phi_1, ..., \phi_m\}|\!\}), B), \omega\omega'\rangle$.

Item (β_{10}) expresses that if $e_i \in \Phi$, then for every execution π s.t. $\pi(0) = \langle p, \gamma_m \cdots \gamma_0\rangle$, $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda \Phi \setminus \{e_i\}$ and $\pi \models^B_\lambda e_i$ (i.e., $(\langle p, \gamma_m \cdots \gamma_0\rangle, B) \in L(e_i)$, meaning there exist $\overrightarrow{S_{m+1}}, ..., \overrightarrow{S_0} \in \overrightarrow{\mathcal{S}}$ s.t. for every j, $0 \leq j \leq m$, $\overrightarrow{S_j}(i) \overset{\gamma_j}{\twoheadrightarrow}_B \overrightarrow{S_{j+1}}(i)$ and $\overrightarrow{S_{m+1}}(i) \in S^i_f$). This is guaranteed by Item (β_{10}) stating $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\{\!|p, \Phi|\!\}), B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$ iff $\mathcal{BP}'_\psi$ has an accepting run from $\langle((\{\!|p, \Phi \setminus \{e_i\}|\!\}), B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$ and there exists $\overrightarrow{S_{m+1}} \in \overrightarrow{\mathcal{S}}$ s.t. $\overrightarrow{S_m}(i) \overset{\gamma_m}{\twoheadrightarrow}_B \overrightarrow{S_{m+1}}(i)$ and $\overrightarrow{S_{m+1}}(i) \in S^i_f$. The intuition behind Item (β_{11}) is similar to Item (β_{10}). Thus, we get that:

Theorem 5. *For every $B \in \mathcal{B}$ and every configuration $\langle p, \gamma_m \cdots \gamma_0\rangle \in P \times \Gamma^*$, $\langle p, \gamma_m \cdots \gamma_0\rangle \models^B_\lambda \psi$ iff there exist $\overrightarrow{S_m}, ..., \overrightarrow{S_0} \in \overrightarrow{\mathcal{S}}$ s.t. $\langle((\{\!|p, \{\psi\}|\!\}), B), (\gamma_m, \overrightarrow{S_m}) \cdots (\gamma_0, \overrightarrow{S_0})\rangle$ is consistent and is in $L(\mathcal{BP}'_\psi)$.*

From Proposition 2, Theorem 1 and Theorem 5, we obtain that:

Theorem 6. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : \mathrm{AP}_\mathcal{D} \to 2^P$ and a SLTPL formula ψ, for every $B \in \mathcal{B}$ and configuration $\langle p, \omega\rangle$, whether or not $\langle p, \omega\rangle$ satisfies ψ under B can be decided in time* $\mathrm{O}(|cl_\mathbf{U}(\psi)| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot |P| \cdot (|\Delta| + |P| \cdot |\Gamma|)^2 \cdot |\overrightarrow{\mathcal{S}}|^2 \cdot 2^{3|\psi|} \cdot |\mathcal{D}|^{3|\mathcal{X}|})$.

The complexity follows from the fact that the number of transition rules (resp. states) of $\mathcal{P}'$ is at most $O(|\Delta| \cdot |\overrightarrow{\mathcal{S}}|)$ (resp. $O(|P|)$) and the stack alphabet $\Gamma' = \Gamma \times \overrightarrow{\mathcal{S}}$. This implies that the number of transition rules (resp. states) of the SBPDS equivalent to $\mathcal{BP}'_\psi$ is at most $O(|cl_\mathbf{U}(\psi)| \cdot (|\Delta| + |P| \cdot |\Gamma|) \cdot |\overrightarrow{\mathcal{S}}| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$ (resp. $O(|P| \cdot |\mathcal{D}| \cdot |\mathcal{X}| \cdot 2^{|\psi|})$).

6 Experiments

We implemented our techniques in a tool for malware detection. We use BDDs to compactly represent the relations Θ. We evaluated our tool on 270 malwares taken from VX Heavens [13] and 27 benign programs taken from Microsoft Windows XP system. All the experiments were run on a Linux platform (Fedora 13) with a 2.4GHz CPU, 2GB of memory. The time limit is fixed to 20 minutes. Moreover, we compared the performances of our techniques with SCTPL [21] and LTL with regular valuations [10] (denoted by LTLr) model-checking. Our tool was able to detect all the malwares. Figures 3 and 4 show the comparison of our techniques with SCTPL and LTLr model-checking for PDSs [10,21]. Due to lack of space, Table 1 shows partial results. The full results can be found in the full version of this paper [22]. Time and memory are given in seconds and MB respectively. **#LOC** denotes the number of instructions of the assembly program. The result *Yes* denotes that the program is detected as a malware, otherwise the result is *No*. As can be seen in Table 1 and Figures 3 and 4, in most cases, SLTPL model-checking performs better. The analysis of several malwares using SCTPL or LTLr model-checking runs out of memory or time, whereas our tool terminates and is able to detect these malwares. Moreover, using

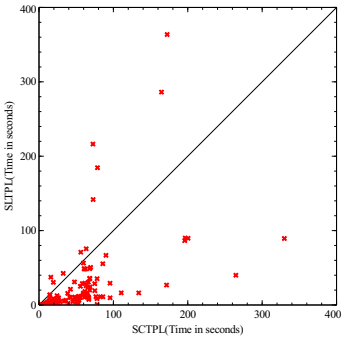

Fig. 3. SLTPL vs. SCTPL

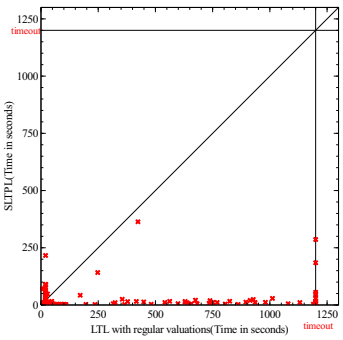

Fig. 4. SLTPL vs. LTLr

Generator	NGVCK	VCL32
No. of Variants	100	100
Our techniques	**100%**	**100%**
Avira	0%	0%
Kaspersky	23%	2%
Avast	18%	100%
Qihoo 360	68%	99%
McAfee	100%	0%
AVG	11%	100%
BitDefender	97%	100%
Eset Nod32	81%	76%
F-Secure	0%	0%
Norton	46%	30%
Panda	0%	0%
Trend Micro	0%	0%

Fig. 5. Detection Ratio.

the SCTPL formula Ψ'_{wv} (described in Section 3.3) causes false alarms when checking 21 benign programs, whereas using SLTPL we correctly classify these programs as benign.

Our tool can also detect the malware *Flame*. Flame is the most complex attack toolkit which can sniff the network traffic, record audio conversations, intercept the keyboard, etc. It has been active for more that 5 years and was not detected by any antivirus.

Moreover, to compare our techniques with the well-known existing anti-viruses, and show the robustness of our tool, we automatically created 200 new malwares using the generators NGVCK and VCL32 [13]. [25] showed that these systems are the best malware generators, compared to the other generators of VX Heavens [13]. These programs use very sophisticated features such as anti-disassembly, anti-debugging, anti-emulation, and anti-behavior blocking and come equipped with code morphing ability which allows them to produce different-looking viruses. Our results are reported in Figure 5. Our techniques were able to detect all these 200 malwares, whereas several well-known and widely used anti-viruses Avira, Avast, Kaspersky, McAfee, AVG, BitDefender, Eset Nod32, F-Secure, Norton, Panda, Trend Micro and Qihoo 360 were not able to detect several of them.

References

1. D. Babic, D. Reynaud, and D. Song. Malware analysis with tree automata inference. In *CAV*, 2011.
2. P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *RV*, pages 168–182, 2010.
3. P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Abstraction-based malware analysis using rewriting and model checking. In *ESORICS*, pages 806–823, 2012.
4. J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *SREIS*, 2001.
5. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, 5:263–270, 2009.
6. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *CONCUR'97*. LNCS 1243, 1997.
7. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, 2003.
8. M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, 2005.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
10. J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput.*, 186(2), 2003.

Table 1. Partial Results of Malware Detection.

	Example	#LOC	SLTPL			SCTPL			LTLr		
			Time	Memory	Result	Time	Memory	Result	Time	Memory	Result
Virus	Akez	264	13.78	59.02	Yes	14.75	15.59	Yes	**timeout**		
	Alcaul.b	904	9.79	37.40	Yes	26.25	1.08	Yes	**timeout**		
	Alcaul.c	347	2.05	9.40	Yes	26.52	2.45	Yes	365.53	225.67	Yes
	Alcaul.d	837	0.24	0.17	Yes	23.52	20.39	Yes	**timeout**		
Email-worm	Kirbster	1261	948.52	1383.02	Yes		**o.o.m.**		**timeout**		
	Krynos.b	18357	987.22	947.92	Yes		**o.o.m.**		**timeout**		
	Newapt.B	11703	1120.21	1042.74	Yes		**o.o.m.**		**timeout**		
	Newapt.F	11771	1045.17	908.35	Yes		**o.o.m.**		**timeout**		
	Newapt.E	11717	1059.45	970.27	Yes		**o.o.m.**		**timeout**		
	Mydoom.j	22335	89.66	40.15	Yes	200.41	48.17	Yes	**timeout**		
	Mydoom.v	5960	10.78	19.03	Yes	66.34	16.49	Yes	1131.00	1010.24	Yes
	Mydoom.y	26902	66.77	36.60	Yes	90.00	43.19	Yes	**timeout**		
Trojan	LdPinch.aar	1245	32.03	198.88	Yes	1.66	8.47	Yes	**timeout**		
	LdPinch.aoq	7688	46.29	234.86	Yes	7.33	10.13	Yes	**timeout**		
	LdPinch.mj	5952	39.07	199.28	Yes	5.74	8.90	Yes	**timeout**		
	LdPinch.ld	6609	8.37	13.36	Yes	5.41	4.24	Yes	452.93	410.85	Yes
Benign	Cmd.exe	35887	109.81	20.00	No		**o.o.m.**		**timeout**		
	Blastcln.exe	13819	103.87	80.53	**No**	27.72	6.30	**Yes**	**timeout**		
	Regsvr32.exe	1280	7.31	26.85	**No**	0.48	1.87	**Yes**	**158.06**	48.15	**No**
	ipv6.exe	13700	89.14	31.04	**No**	60.45	3.14	**Yes**	**timeout**		
	dplaysvr.exe	6796	35.46	30.39	**No**	17.12	2.84	**Yes**	**timeout**		
	Shutdown.exe	2524	31.69	62.93	No		**o.o.m.**		**timeout**		
	Regedt.exe	60	0.02	0.02	**No**	10.62	0.03	**Yes**	0.02	0.02	**No**
	Java.exe	21868	184.58	27.96	**No**	78.64	238.77	**Yes**	**timeout**		

11. J. Esparza and S. Schwoon. A BDD-Based Model Checker for Recursive Programs. In *CAV*, 2001.
12. O. Grumberg, O. Kupferman, and S. Sheinvald. Variable Automata over Infinite Alphabets. In *LATA*, pages 561–572, 2010.
13. V. Heavens. http://vx.netlux.org.
14. I. M. Hodkinson, F. Wolter, and M. Zakharyaschev. Monodic fragments of first-order temporal logics. In *LPAR*, pages 1–23, 2001.
15. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *DIMVA*, 2005.
16. O. Kupferman, N. Piterman, and M. Y. Vardi. An automata-theoretic approach to infinite-state systems. In *Essays in Memory of Amir Pnueli*, 2010.
17. A. Lakhotia, D. R. Boccardo, A. Singh, and A. Manacero. Context-sensitive analysis of obfuscated x86 executables. In *PEPM*, 2010.
18. A. Lakhotia, E. U. Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Trans. Software Eng.*, 31(11), 2005.
19. P. K. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *IAW*, 2003.
20. A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for büchi automata with appplications to temporal logic. *Theor. Comput. Sci.*, 49:217–237, 1987.
21. F. Song and T. Touili. Efficient malware detection using model-checking. In *FM*, 2012.
22. F. Song and T. Touili. LTL Model-Checking for Malware Detection. Technical report, LIAFA, CNRS, `http://www.liafa.univ-paris-diderot.fr/~song/SLTPL.pdf`, 2012.
23. F. Song and T. Touili. Pushdown model checking for malware detection. In *TACAS*, 2012.
24. F. Wang, S. Tahar, and O. A. Mohamed. First-Order LTL Model Checking Using MDGs. In *ATVA*, 2004.
25. W. Wong. Analysis and detection of metamorphic computer viruses. Master's thesis, San Jose State University, 2006.
26. Y. Xu, E. Cerny, X. Song, F. Corella, and O. A. Mohamed. Model checking for a first-order temporal logic using multiway decision graphs. In *CAV*, 1998.