CARET Model Checking for Malware Detection

Huu-Vu Nguyen
University Paris Diderot and LIPN, France

Tayssir Touili
LIPN, CNRS and University Paris 13, France

ABSTRACT

The number of malware is growing significantly fast. Traditional malware detectors based on signature matching or code emulation are easy to get around. To overcome this problem, model-checking emerges as a technique that has been extensively applied for malware detection recently. Pushdown systems were proposed as a natural model for programs, since they allow to keep track of the stack, while extensions of LTL and CTL were considered for malicious behavior specification. However, LTL and CTL like formulas don't allow to express behaviors with matching calls and returns. In this paper, we propose to use CARET for malicious behavior specification. Since CARET formulas for malicious behaviors are huge, we propose to extend CARET with variables, quantifiers and predicates over the stack. Our new logic is called SPCARET. We reduce the malware detection problem to the model checking problem of PDSs against SPCARET formulas, and we propose efficient algorithms to model check SPCARET formulas for PDSs. We implemented our algorithms in a tool for malware detection. We obtained encouraging results.

KEYWORDS

Pushdown systems, CARET formulas, model checking, malware detection

ACM Reference format:
Huu-Vu Nguyen and Tayssir Touili. 2017. CARET Model Checking for Malware Detection. In *Proceedings of , USA, 2017 (SPIN'2017),* 10 pages.
DOI: 10.475/123_4

1 INTRODUCTION

The number of malware is growing fast recently. Traditional malware detection techniques including signature matching and code emulation are not efficient enough. While malware writers can use obfuscation techniques to bypass the signature based malware detectors easily, code emulation can only monitor programs in certain execution paths during a short time. To overcome these limitations, model-checking emerges as an efficient technique for malware detection, as model-checking allows to check the behaviours of a program in all its execution traces without executing it.

A lof of efforts have been made to apply model-checking for malware detection [2, 4, 7, 11, 13–15]. In [7], the authors proposed to use finite state graphs to model the program and use the temporal logic CTPL to describe malicious behaviours. However, finite graphs are not precise enough to model programs, as they don't allow to keep track of the program's stack.

Being able to record the program's stack is very important for malware detection as explained in [9]. Indeed, *push* and *pop* instructions are frequently used by malware writers for code obfuscation. Moreover, in binary codes and assembly programs, parameters are passed to functions via the stack (i.e., they are pushed on the stack before invoking the function). The values of these parameters determine whether the program has a malicious behavior or not. Let us consider the example of an email worm to illustrate this. The typical behaviour of an email worm is to copy itself to different locations. To do this, the email worm first calls the API *GetModuleFileNameA* with 0 and the memory address d as parameters (i.e., 0 and d on top of the stack). Then, it calls *CopyFileA* with the same d as parameter (i.e., with d on the top of the stack). *GetModuleFileNameA* will put the file name of the current executable in the memory address d. After that, $CopyFileA$ will use the file name stored at the memory address d to copy itself to other locations. So, to determine whether a program that calls the API functions *GetModuleFileNameA* and *CopyFileA* successively is a malware or not, it is important to check whether the top of the stack contains $0d$ for the first call, and d for the second. Thus, it is very important to track the program's stack for malware detection. To this aim, [13–16] proposed to use pushdown systems to model programs, and defined extensions of LTL and CTL (called SLTPL and SCTPL) to precisely and succinctly represent malicious behaviors. However, these logics cannot describe properties that require matchings of calls and returns, which is necessary to specify malicious behaviours. For instance, let us consider a typical behaviour of a spyware: it consists in hunting for personal information (emails, bank account information, ...) on local drives by searching files matching certain conditions. To do this, the spyware first calls the API function *FindFirstFileA* to obtain the first matching file. *FindFirstFileA* will return a search handle h. To obtain all matching files, the spyware must continuously call the function *FindNextFileA* with h as parameter. This behaviour can not be specified by LTL or CTL since it requires that the return value of the API *FindFirstFileA* must be used as the input of the function *FindNextFileA*.

CARET was introduced to express these properties that involve matchings of calls and returns[1]. Using CARET, the above behavior can be expressed by the following formula:

$$\psi_{sf} = \bigvee_{d \in D} F^g(\text{call(FindFirstFileA)} \land X^a(eax = d) \land F^a(\text{call(FindNextFileA)} \land d\Gamma^*))$$

where the $\bigvee$ is taken over all possible memory addresses d which contain the values of search handles h in the program. F^g is the standard LTL F operator (in the future), while, roughly speaking, F^a is a CARET operator that means "in the future after the return point of the current procedure", and X^a means "at the return point of the current procedure".

In binary codes and assembly programs, the return value of an API function is put in the register eax. Thus, the return value of $FindFirstFileA$ is the value of eax at its corresponding return-point. Then, the subformula F^g(call(FindFirstFileA) $\land X^a(eax =$

d)) states that there is a call to the API *FindFirstFileA* and the return value of this function is d. When *FindNextFileA* is invoked, it requires a search handle as parameter. The requirement that d is on top of the program stack is expressed by the regular expression $d\Gamma^*$. Thus, the subformula call(FindNextFileA) $\wedge\, d\Gamma^*$ expresses that *FindNextFileA* is called with d as parameter (d stores the information of the search handle). ψ_{sf} expresses then that there is a call to the API *FindFirstFileA* with the return value d (the search handle), followed by a call to the function *FindNextFileA* with d on the top of the stack.

However, this formula is huge, as it considers the disjunction (of different CARET formulas) over all possible memory addresses d which contain the information of search handles h in the program. To represent it in a more succinct fashion, we follow the idea of [7, 14, 15] and extend CARET with variables, quantifiers, and predicates over the stack. We call our new logic SPCARET. We define also PCARET formulas to be SPCARET formulas that do not use predicates over the stack. The above formula can be compactly represented in SPCARET as follows:

$$\psi_{sf2} = \exists \mathbf{x} F^g(\text{call(FindFirstFileA)} \wedge X^a(eax = x) \wedge F^a(\text{call(FindNextFileA)} \wedge x\Gamma^*))$$

Thus, we propose in this work to use pushdown systems (PDSs) to model programs, and SPCARET formulas to specify malicious behaviors. We reduce the malware detection problem to the model checking problem of PDSs against SPCARET formulas, and we propose efficient algorithms to model check PCARET and SPCARET formulas for PDSs. Our algorithms are based on reducing the model checking problem to the emptiness problem of Symbolic Büchi Pushdown Systems. This latter problem is already solved in [3, 5]. We implemented our techniques in a tool for malware detection. We obtained encouraging results. Our tool was able to detect several malwares and to determine that benign programs are benign. We compared the performance of our tool against a standard CARET model checking tool. Our tool behaves much better, as the standard CARET tool timeout in most cases.

The rest of paper is organized as follows. In Section 2, we recall the definitions of Pushdown Systems. Section 3 introduces our logic. Model checking PCARET and SPCARET are provided in Sections 4, 5 and 6. In Section 7, we present our experimental results. Due to lack of space, the proofs can be found in the full version of the paper [10].

2 BINARY CODE MODELLING

We use Pushdown Systems (PDSs) to model binary programs. Indeed, PDSs are known to be a natural model for sequential programs [12]. We apply the translation of [13] together with the tools [6, 8] to obtain Pushdown Systems from binary programs. As will be discussed in the next section, to precisely describe malicious behaviors, we need to keep track of the call and return actions in each path. Thus, we adapt the PDS model in order to record whether a rule of a PDS corresponds to a *call*, a *return*, or another instruction. We call this model a *Labelled Pushdown System.* We also extend the notion of *path* in order to take into account matching returns of calls.

2.1 Labelled Pushdown Systems

Definition 2.1. *A Labelled Pushdown System (PDS) $\mathcal{P}$ is a tuple (P, Γ, Δ), where P is a finite set of control locations, Γ is a finite set of stack alphabets, and Δ is a finite subset of $((P \times \Gamma) \times (P \times \Gamma^*) \times \{call, ret, int\})$. If $((p, \gamma), (q, \omega), t) \in \Delta$ (where $t \in \{call, ret, int\}$), we also write $\langle p, \gamma \rangle \xrightarrow{t} \langle q, \omega \rangle \in \Delta$. Transition rules of Δ are in the following form, in which $p \in P, q \in P, \gamma, \gamma_1, \gamma_2 \in \Gamma$, and $\omega \in \Gamma^*$:*

- *(r_1): $\langle p, \gamma \rangle \xrightarrow{call} \langle q, \gamma_1\gamma_2 \rangle$*
- *(r_2): $\langle p, \gamma \rangle \xrightarrow{ret} \langle q, \epsilon \rangle$*
- *(r_3): $\langle p, \gamma \rangle \xrightarrow{int} \langle q, \omega \rangle$*

Roughly speaking, a transition rule in the form $\langle p, \gamma \rangle \xrightarrow{call} \langle q, \gamma_1\gamma_2 \rangle$ describes a call statement. This rule usually models a statement in the form $\gamma \xrightarrow{call\ proc} \gamma_2$. In this rule, γ corresponds to the control point of the program in which the function call is invoked, γ_1 is the entry point of the invoked procedure, and γ_2 is the return point of that call. A rule r_2 represents a return statement, while a rule r_3 models a *simple* statement. In the remainder of this paper, we will also refer to Labelled Pushdown Systems as Pushdown Systems, since there is no ambiguity.

Let $\langle p, \omega \rangle$, where $p \in P$ and $\omega \in \Gamma^*$ be a configuration of $\mathcal{P}$. The transition relation $\Rightarrow_{\mathcal{P}}$ is defined as follows: If $\langle p, \gamma \rangle \xrightarrow{t} \langle q, \omega \rangle$, then for every $\omega' \in \Gamma^*$, $\langle p, \gamma\omega' \rangle \Rightarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$. In other words, $\langle q, \omega\omega' \rangle$ is an immediate successor of $\langle p, \gamma\omega' \rangle$. An execution of $\mathcal{P}$ starting from $\langle p_0, \omega_0 \rangle$ is an infinite sequence of configurations $\langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle$... where $\langle p_i, \omega_i \rangle \in P \times \Gamma^*$ and $\langle p_{i+1}, \omega_{i+1} \rangle$ is an immediate successor of $\langle p_i, \omega_i \rangle$ for every $i \geq 0$.

Let $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle$... be an execution of $\mathcal{P}$. We associate to each configuration $\langle p_i, \omega_i \rangle$ of π a tag $t_i(\pi)$ in $\{call, int, ret\}$ as follows:

- If $\langle p_i, \omega_i \rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1}, \omega_{i+1} \rangle$ corresponds to a call statement, then $t_i(\pi) = call$.
- If $\langle p_i, \omega_i \rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1}, \omega_{i+1} \rangle$ corresponds to a return statement, then $t_i(\pi) = ret$.
- If $\langle p_i, \omega_i \rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1}, \omega_{i+1} \rangle$ corresponds to a simple statement (neither a call statement nor a return statement), then $t_i(\pi) = int$.

2.2 Global, abstract and caller paths

Let $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle$... be a path of PDS. Over π, three kinds of successors are defined for every position $\langle p_i, \omega_i \rangle$:

- *global-successor*: The global-successor of $\langle p_i, \omega_i \rangle$ is $\langle p_{i+1}, \omega_{i+1} \rangle$.
- *abstract-successor*: The abstract-successor of $\langle p_i, \omega_i \rangle$ is determined by its associated tag $t_i(\pi)$.
  - If $t_i(\pi) = call$, the abstract successor of $\langle p_i, \omega_i \rangle$ is the matching return point.
  - If $t_i(\pi) = int$, the abstract-successor of $\langle p_i, \omega_i \rangle$ is $\langle p_{i+1}, \omega_{i+1} \rangle$.
  - If $t_i(\pi) = ret$, the abstract successor of $\langle p_i, \omega_i \rangle$ is defined as $\perp$.
- *caller-successor*: The caller-successor of $\langle p_i, \omega_i \rangle$ is the most inner unmatched call if there is such a *call.* Otherwise, it is defined as $\perp$.

For example, in Figure 1:

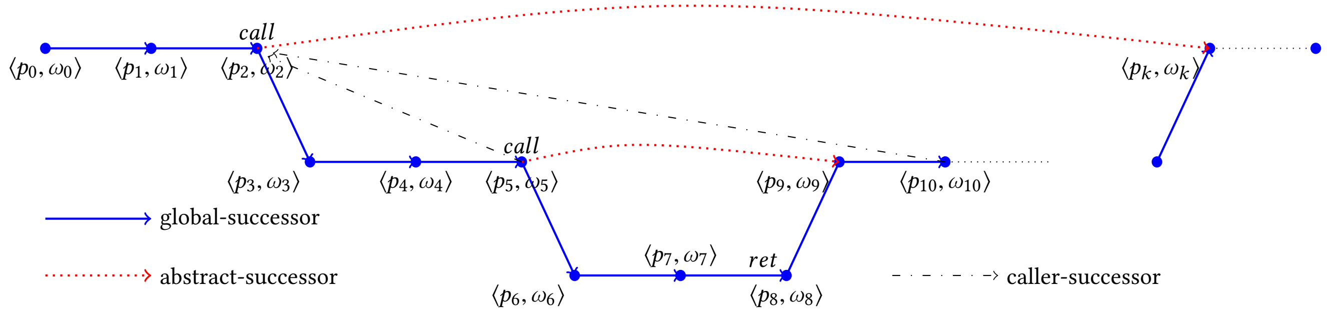

call
$\langle p_0, \omega_0 \rangle$
$\langle p_1, \omega_1 \rangle$
$\langle p_2, \omega_2 \rangle$
$\langle p_k, \omega_k \rangle$
call
$\langle p_3, \omega_3 \rangle$
$\langle p_4, \omega_4 \rangle$
$\langle p_5, \omega_5 \rangle$
$\langle p_9, \omega_9 \rangle$
$\langle p_{10}, \omega_{10} \rangle$
global-successor
$\langle p_7, \omega_7 \rangle$
ret
abstract-successor
$\langle p_6, \omega_6 \rangle$
$\langle p_8, \omega_8 \rangle$
caller-successor

Figure 1: Three kinds of successors of CARET

- The global-successor of $\langle p_1, \omega_1 \rangle$ and $\langle p_2, \omega_2 \rangle$ are $\langle p_2, \omega_2 \rangle$ and $\langle p_3, \omega_3 \rangle$ respectively.
- The abstract-successor of $\langle p_2, \omega_2 \rangle$ and $\langle p_5, \omega_5 \rangle$ are $\langle p_k, \omega_k \rangle$ and $\langle p_9, \omega_9 \rangle$ respectively.
- The caller-successor of $\langle p_6, \omega_6 \rangle$, $\langle p_7, \omega_7 \rangle$, $\langle p_8, \omega_8 \rangle$ is $\langle p_5, \omega_5 \rangle$ while the caller-successor of $\langle p_3, \omega_3 \rangle$, $\langle p_4, \omega_4 \rangle$, $\langle p_5, \omega_5 \rangle$, $\langle p_9, \omega_9 \rangle$ is $\langle p_2, \omega_2 \rangle$. Note that the caller-successor of $\langle p_0, \omega_0 \rangle$, $\langle p_1, \omega_1 \rangle$, $\langle p_2, \omega_2 \rangle$, $\langle p_k, \omega_k \rangle$ is $\perp$.

A *global-path* is obtained by applying repeatedly the global-successor operator. Similarly, an *abstract-path* or a *caller-path* are obtained by repeatedly applying the abstract-successor and caller-successor respectively. In Figure 1, from $\langle p_4, \omega_4 \rangle$, the global-path is $\langle p_4, \omega_4 \rangle \langle p_5, \omega_5 \rangle \langle p_6, \omega_6 \rangle \langle p_7, \omega_7 \rangle \langle p_8, \omega_8 \rangle \langle p_9, \omega_9 \rangle \langle p_{10}, \omega_{10} \rangle \ldots$, the abstract-path is $\langle p_4, \omega_4 \rangle \langle p_5, \omega_5 \rangle \langle p_9, \omega_9 \rangle \langle p_{10}, \omega_{10} \rangle \ldots$ while the caller-path is $\langle p_4, \omega_4 \rangle \langle p_2, \omega_2 \rangle$. Note that the caller-path is always finite.

3 MALICIOUS BEHAVIOUR SPECIFICATION

In this section, we define the Stack linear temporal Predicate logic of CAlls and RETurns (SPCARET) as an extension of the linear temporal logic of CAlls and RETurns (CARET) with variables and regular predicates over the stack contents. The predicates contain variables that can be quantified existentially or universally. Regular predicates are expressed by regular variable expressions and are used to describe the stack content of PDSs.

3.1 Environments, Predicates and Regular Variable Expressions

Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a finite set of variables over a finite domain $\mathcal{D}$. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment that associates each variable $x \in \mathcal{X}$ with a value $d \in \mathcal{D}$ s.t $B(d) = d$ for every $d \in \mathcal{D}$. Let $B[x \leftarrow d]$ be an environment obtained from B such that $B[x \leftarrow d](x) = d$ and $B[x \leftarrow d](y) = B(y)$ for every $y \neq x$. Let $\mathcal{B}$ be the set of all environments. Let $\theta_{id} = \{(B, B') \in \mathcal{B} \times \mathcal{B} \mid B = B'\}$ be the identity relation for environments, and for $x \in \mathcal{X}$, let $\theta_x = \{(B, B') \in \mathcal{B} \times \mathcal{B} \mid \forall y \in \mathcal{X}, y \neq x, B(y) = B'(y)\}$ be the relation that abstracts away the value of x.

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

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a Labelled PDS. A Regular Variable Expression (RVE) e over $\mathcal{X} \cup \Gamma$ is defined by $e ::= \epsilon \mid a \in \mathcal{X} \cup \Gamma \mid e + e \mid e.e \mid e^*$. The language $L(e)$ of a RVE e is a subset of $P \times \Gamma^* \times \mathcal{B}$ and is defined as follows:

- $L(\epsilon) = \{(\langle p, \epsilon \rangle, B) \mid p \in P, B \in \mathcal{B}\}$
- for $x \in \mathcal{X}$, $L(x) = \{(\langle p, \gamma \rangle, B) \mid p \in P, \gamma \in \Gamma, B \in \mathcal{B} \text{ s.t } B(x) = \gamma\}$
- for $\gamma \in \Gamma$, $L(\gamma) = \{(\langle p, \gamma \rangle, B) \mid p \in P, B \in \mathcal{B}\}$
- $L(e_1.e_2) = \{(\langle p, \omega' \omega'' \rangle, B) \mid (\langle p, \omega' \rangle, B) \in L(e_1); (\langle p, \omega'' \rangle, B) \in L(e_2)\}$
- $L(e^*) = \{(\langle p, \omega \rangle, B) \mid \omega \in \{v \in \Gamma^* \mid (\langle p, v \rangle, B) \in L(e)\}^*\}$

3.2 The Stack linear temporal Predicate logic of CAlls and RETurns - SPCARET

A SPCARET formula is a CARET [1] formula where predicates and RVEs are used as atomic propositions and where quantifiers are applied to variables. For technical reasons, we assume w.l.o.g. that formulas are written in positive normal form, where negations are applied only to atomic predicates, and we use the *release operator* R as the dual of the until operator U. From now on, we fix a finite set of variables $\mathcal{X}$, a finite set of atomic propositions AP, a finite domain $\mathcal{D}$, and a finite set of RVEs $\mathcal{V}$. A SPCARET formula is defined as follows, where $v \in \{g, a, c\}$, $x \in \mathcal{X}$, $e \in \mathcal{V}$, $b(\alpha_1, \ldots, \alpha_n) \in AP_{\mathcal{X}}$:

$$\psi := b(\alpha_1, \ldots, \alpha_n) \mid \neg b(\alpha_1, \ldots, \alpha_n) \mid e \mid \neg e \mid \psi \vee \psi \mid \psi \wedge \psi \mid \forall x \psi \mid \exists x \psi \mid X^v \psi \mid \psi U^v \psi \mid \psi R^v \psi$$

Let $\lambda : P \longrightarrow 2^{AP_{\mathcal{D}}}$ be a labelling function which associates each control location to a set of atomic predicates. Let ψ be a SPCARET formula over AP. Let $\langle p, \omega \rangle$ be a configuration of $\mathcal{P}$. Then we say that $\mathcal{P}$ satisfies ψ at $\langle p, \omega \rangle$ (denoted by $\langle p, \omega \rangle \models_\lambda \psi$) iff there exists an environment $B \in \mathcal{B}$, a path $\pi = \langle p_0, \omega_0 \rangle \langle p_1, \omega_1 \rangle \ldots$ starting from $\langle p, \omega \rangle$ such that π satisfies ψ under B (denoted by $\pi \models_\lambda^B \psi$). Let $next_i^g$, $next_i^a$ and $next_i^c$ be the global-successor, abstract-successor and caller-successor of $\langle p_i, \omega_i \rangle$ respectively. Let (π, i) be the suffix of π starting from $\langle p_i, \omega_i \rangle$. Then, $\pi \models_\lambda^B \psi$ iff $(\pi, 0) \models_\lambda^B \psi$ where $(\pi, i) \models_\lambda^B \psi$ is defined inductively as follows:

- $(\pi, i) \models_\lambda^B b(\alpha_1, \ldots, \alpha_n)$, iff $b(B(\alpha_1), \ldots, B(\alpha_n)) \in \lambda(p_i)$
- $(\pi, i) \models_\lambda^B \neg b(\alpha_1, \ldots, \alpha_n)$, iff $b(B(\alpha_1), \ldots, B(\alpha_n)) \notin \lambda(p_i)$
- $(\pi, i) \models_\lambda^B e$ iff $(\langle p_i, \omega_i \rangle, B) \in L(e)$
- $(\pi, i) \models_\lambda^B \neg e$ iff $(\langle p_i, \omega_i \rangle, B) \notin L(e)$
- $(\pi, i) \models_\lambda^B \psi_1 \vee \psi_2$ iff $((\pi, i) \models_\lambda^B \psi_1$ or $(\pi, i) \models_\lambda^B \psi_2)$
- $(\pi, i) \models_\lambda^B \psi_1 \wedge \psi_2$ iff $((\pi, i) \models_\lambda^B \psi_1$ and $(\pi, i) \models_\lambda^B \psi_2)$

- $(\pi, i) \models^B_\lambda X^g \psi$ iff $(\pi, next^g_i) \models^B_\lambda \psi$
- $(\pi, i) \models^B_\lambda X^a \psi$ iff $next^a_i \neq \perp$ and $(\pi, next^a_i) \models^B_\lambda \psi$
- $(\pi, i) \models^B_\lambda X^c \psi$ iff $next^c_i \neq \perp$ and $(\pi, next^c_i) \models^B_\lambda \psi$
- $(\pi, i) \models^B_\lambda \forall x \psi$ iff for every $d \in \mathcal{D}$, $(\pi, i) \models^{B[x \leftarrow d]}_\lambda \psi$
- $(\pi, i) \models^B_\lambda \exists x \psi$ iff there exists $d \in \mathcal{D}$, $(\pi, i) \models^{B[x \leftarrow d]}_\lambda \psi$
- $(\pi, i) \models^B_\lambda \psi_1 U^v \psi_2$ (with $v \in \{g, a, c\}$) iff there exists a sequence of positions $h_0, h_1, ..., h_{k-1}, h_k$ where $h_0 = i$, for every $0 \leq j \leq k-1$: $h_{j+1} = next^v_{h_j}, (\pi, h_j) \models^B_\lambda \psi_1$ and $(\pi, h_k) \models^B_\lambda \psi_2$
- $(\pi, i) \models^B_\lambda \psi_1 R^v \psi_2$ (with $v \in \{g, a, c\}$) iff there exists a sequence of positions $h_0, h_1, ..., h_{k-1}, h_k$ where $h_0 = i$, for every $0 \leq j \leq k$: $h_{j+1} = next^v_{h_j}, (\pi, h_j) \models^B_\lambda \psi_2$ and $(\pi, h_k) \models^B_\lambda \psi_1$

Other CARET operators can be represented by the above operators: $F^g \psi$ = true $U^g \psi$, $G^g \psi$ = false $R^g \psi$, $F^a \psi$ = true $U^a \psi$, $G^a \psi$ = false $R^a \psi$, $F^c \psi$ = true $U^c \psi$, $G^c \psi$ = false $R^c \psi$,....

Let a PCARET formula be an SPCARET formula that does not use any regular variable expression.

CARET with regular valuations is an extension of CARET where the set of configurations where an atomic proposition hold can be expressed by a regular language [11]. Since the domain $\mathcal{D}$ is finite, we get:

Proposition 3.1. *PCARET and CARET (resp. SPCARET and CARET with regular valuations) have the same expressive power. SPCARET is more expressive than CARET.*

Let ψ be a SPCARET formula. The closure of ψ, denoted $Cl(\psi)$, is the smallest set that contains ψ and satisfies the following properties:

- if $\neg\psi' \in Cl(\psi)$, then $\psi' \in Cl(\psi)$
- if $X^v \psi' \in Cl(\psi)$ (with $v \in \{g, a, c\}$), then $\psi' \in Cl(\psi)$
- if $\forall x \psi' \in Cl(\psi)$, then, $\psi' \in Cl(\psi)$ and for every $d \in \mathcal{D}$, $\psi'_d \in Cl(\psi)$ where ψ'_d is ψ' in which x is substituted by d
- if $\exists x \psi' \in Cl(\psi)$, then, $\psi' \in Cl(\psi)$ and for every $d \in \mathcal{D}$, $\psi'_d \in Cl(\psi)$ where ψ'_d is ψ' in which x is substituted by d
- if $\psi_1 \vee \psi_2 \in Cl(\psi)$, then $\psi_1 \in Cl(\psi), \psi_2 \in Cl(\psi)$
- if $\psi_1 \wedge \psi_2 \in Cl(\psi)$, then $\psi_1 \in Cl(\psi), \psi_2 \in Cl(\psi)$
- if $\psi_1 U^v \psi_2 \in Cl(\psi)$ (with $v \in \{g, a, c\}$), then $\psi_1 \in Cl(\psi), \psi_2 \in Cl(\psi), X^v(\psi_1 U^v \psi_2) \in Cl(\psi)$
- if $\psi_1 R^v \psi_2 \in Cl(\psi)$ (with $v \in \{g, a, c\}$), then $\psi_1 \in Cl(\psi), \psi_2 \in Cl(\psi), X^v(\psi_1 R^v \psi_2) \in Cl(\psi)$
- if $\psi' \in Cl(\psi)$, and ψ' is not in the form $\neg\psi''$ then $\neg\psi' \in Cl(\psi)$

3.3 Modelling Malicious Behaviours Using SPCARET

We showed in the introduction how we can use our new logic to describe the malicious behavior of a spyware. In this section, we show how SPCARET can be used to succinctly specify different other malicious behaviours.

Open and listen on a specific port: Malware writers usually set up the malware to listen to a certain port to get updated information (new attack targets, ...). To achieve this task, it needs to call the API *socket* to create a socket, followed by a call to the API *bind* to associate a local address with the socket and a call to *listen* to put the socket in the listening state. The call to the API *socket* returns a descriptor referencing the new socket which is used as input of the calls to the APIs *bind* and *listen*. Thus, when *bind* and *listen* are invoked, the socket descriptor must be on top of the program's stack. This kind of malicious behaviours cannot be described by LTL or CTL since it requires the return value of the API function *socket* to be used as the input of the functions *bind* and *listen*. Using SPCARET, this malicious behaviour can be specified as follows:

$$\psi_{lp} = \exists x F^g(\text{call(socket)} \wedge X^a(eax = x) \wedge F^a(\text{call(bind)} \wedge x\Gamma^* \wedge F^a(\text{call(listen)} \wedge x\Gamma^*)))$$

Note that the return value of an API function is put in *eax* when the function terminates. Thus, the return value of an API function is the value of *eax* at its return-point. Then, the subformula $call(socket) \wedge X^a(eax = x)$ states that there is a call to the API function *socket* whose return value is x (since the return-point of a call is its abstract successor). When *bind* is invoked, one required parameter is the socket descriptor and this descriptor must be put on top of the stack (since parameters are passed via the stack in binary programs). The regular variable expression $x\Gamma^*$ describes the requirement that x is on top of the stack. Then, the subformulas $call(bind) \wedge x\Gamma^*$ and $call(listen) \wedge x\Gamma^*$ state that there are calls to *bind* and *listen* whose socket descriptor is x. Thus, ψ_{lp} expresses that there is a call to the API *socket* with a return value x, followed by a call to the function *bind* and a call to the function *listen* with x on top of the stack. Note that in this case, x is the memory address storing the descriptor.

Registry Key Injecting: One typical behaviour of a malware is to add its own executable name to the registry listing so that it can be started at the boot time. To do this, the malware need to invoke *GetModuleFileNameA* with 0 and x as parameters, where x is the memory address that contains the file name of the current executable. After that, the malware calls *RegSetValueExA* with the same x as parameter. *RegSetValueExA* will use the file name stored at x to add itself into the registry key listing. This malicious behaviour can be specified by SPCARET as follows:

$$\psi_{ki} = \exists x F^g(call(GetModuleFileNameA) \wedge 0x\Gamma^* \wedge F^a(call(RegSetValueExA) \wedge x\Gamma^*))$$

This formula states that there is a call to the API *GetModuleFileNameA* with 0 and x on the top of the stack (i.e., with 0 and x as parameters), followed by a call to the API *RegSetValueExA* with x on the top of the stack.

Email Worm: The typical behaviour of an email worm is to infect other files by copying itself to other locations. This is obtained by first calling the API function *GetModuleFileNameA* with 0 and x on top of the program stack (0 and x are parameters), followed by a call to *CopyFileA* with x as parameter. When *GetModuleFileNameA* is executed, the memory address x will contain the file name of the current executable. Then, the call to *CopyFileA* with x on top of the stack will allow the email worm copy itself to another location. Using SPCARET, this behavior is described as follows:

$$\psi_{em} = \exists x F^g(\text{call(GetModuleFileNameA)} \wedge 0x\Gamma^* \wedge F^a(\text{call(CopyFileA)} \wedge x\Gamma^*))$$

This formula states that there is a call to the API function *GetModuleFileNameA* with 0 and x on the top of the stack, followed by a call to the API *CopyFileA* with the same x on the top of stack.

4 SPCARET MODEL-CHECKING FOR PUSHDOWN SYSTEMS

4.1 Using CARET Model-Checking

We can show that:

Theorem 4.1. *Model-checking a PCARET formula against PDSs can be reduced to model-checking a CARET formula against PDSs. Model-checking a SPCARET formula against PDSs can be reduced to model-checking a CARET formula with regular valuations against PDSs.*

The reduction underlying this theorem is based on enumerating all possible values for each variable that occurs in the given SPCARET (PCARET) formula. For example, the PCARET formula $\psi = \exists x_1 \exists x_2\ push(x_1) \wedge X^g push(x_2)$ where x_1 and x_2 are variables over the domain $\mathcal{D} = \{eax, ebx, ecx, ...\}$ can be rewritten as the huge CARET formula $\bigvee_{d_1, d_2 \in \mathcal{D}} push(d_1) \wedge X^g push(d_2)$.

More precisely, we get:

Proposition 4.1. *Let ψ be a SPCARET formula, let $|\mathcal{X}|$ be the number of variables in ψ, let $\mathcal{D}$ be the domain of variables, we can compute an equivalent CARET formula with regular valuations ψ' such that $|\psi'| = |\psi| \times O(|\mathcal{D}|^{|\mathcal{X}|})$.*

CARET model checking for PDSs was solved in [11] :

Theorem 4.2. *[11] Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : P \longrightarrow 2^{AP}$ and a CARET formula ψ, for every configuration $\langle p, \omega \rangle$, whether or not $\langle p, \omega \rangle$ satisfies ψ can be solved in time $|P|.|\Delta|^2.|\Gamma|^2.2^{O(|\psi|)}$.*

Thus, from Theorem 4.2 and Proposition 4.1, we get:

Theorem 4.3. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : P \longrightarrow 2^{AP_{\mathcal{D}}}$ and a SPCARET formula ψ, for every configuration $\langle p, \omega \rangle$, checking whether $\langle p, \omega \rangle$ satisfies ψ by translating ψ to an equivalent CARET formula ψ' can be solved in time $|P|.|\Delta|^2.|\Gamma|^2.2^{O(|\psi||\mathcal{D}|^{|\mathcal{X}|})}$.*

4.2 SPCARET$^{\setminus c}$

It is obvious to see that the above approach that consists in translating a SPCARET formula to an equivalent CARET formula is not efficient since the size of the domain $\mathcal{D}$ is big when the formula specifies a malicious behavior, where $\mathcal{D}$ is usually all possible register names, or all possible values of the memory addresses or all possible values of the stack. Thus, we need a direct model checking algorithm that does not go through the translation to CARET and that is more efficient than the above approach. One possible idea to have a direct algorithm consists in reducing the model checking problem to the emptiness problem of Symbolic Büchi Pushdown Systems (SBPDSs), by computing a kind of product between the SPCARET formula ψ and the PDS $\mathcal{P}$, which gives a Symbolic Büchi Pushdown System. The key idea would be the use of *Symbolic* BPDSs. This allows to move the complexity of dealing with variables over a big domain to the *symbolic* transitions of the BPDS, which can be efficiently dealt with using BDDs as described in [5].

Intuitively, when computing the product, each state of the computed Symbolic BPDS ensures the satisfiability of a certain subformula at some state of the PDS. To be able to apply this approach in the presence of variables, the semantic correctness of a certain subformula at one state is ensured by the semantic correctness of the formulas of its *successor* state. However, this cannot apply for caller-paths since from a state, the correctness of X^c, U^c, and R^c are ensured backward not forward (i.e., by looking at the predecessors, not the successors). Thus, to be able to apply this idea, we define a subclass of SPCARET that does not involve the X^c, U^c, and R^c operators. This subclass is called SPCARET$^{\setminus c}$:

Definition 4.1. *A SPCARET$^{\setminus c}$ (PCARET$^{\setminus c}$) formula is a SPCARET (PCARET) formula that does not use the operators X^c, U^c, and R^c.*

We believe that these operators X^c, U^c, and R^c are not useful to specify malicious behaviours. Indeed, a malicous behaviour can often be described as a sequence of API function calls with corresponding register as well as stack values at calls and matching return-points, combined with a sequence of certain assembly instructions (mov, push, pop,...). The operators X^g, U^g, R^g, X^a, U^a, and R^a are sufficient to express such behaviors.

5 PCARET$^{\setminus c}$ MODEL-CHECKING FOR PUSHDOWN SYSTEMS

In this section, we show how to reduce PDSs model-checking for PCARET$^{\setminus c}$ to the emptiness problem of Symbolic Büchi Pushdown Systems. The latter problem is already solved in [5].

5.1 Symbolic Büchi Pushdown Systems

Definition 5.1. *A Symbolic Pushdown System (SPDS) $\mathcal{P}$ is a tuple (P, Γ, Δ) where P is a finite set of control locations, Γ is a finite set of stack alphabet and Δ is a finite set of symbolic transition rules in the form $\langle p, \gamma \rangle \xrightarrow{\theta} \langle q, \omega \rangle$ where $p, q \in \mathcal{P}, \gamma \in \Gamma, \omega \in \Gamma^*$ and $\theta \subseteq \mathcal{B} \times \mathcal{B}$.*

A symbolic transition rule $\langle p, \gamma \rangle \xrightarrow{\theta} \langle q, \omega \rangle$ represents the set of transition rules: $\langle (p, B), \gamma \rangle \longrightarrow \langle (q, B'), \omega \rangle$ such that $B, B' \in \mathcal{B}$ and $(B, B') \in \theta$. For every $\omega' \in \Gamma'$, $\langle (q, B'), \omega\omega' \rangle$ is an immediate successor of $\langle (p, B), \gamma\omega' \rangle$. A run of $\mathcal{P}$ starting from $\langle (p_0, B_0), \omega_0 \rangle$ is a sequence $\langle (p_0, B_0), \omega_0 \rangle\ \langle (p_1, B_1), \omega_1 \rangle$ ... s.t. $\langle (p_{i+1}, B_{i+1}), \omega_{i+1} \rangle$ is an immediate successor of $\langle (p_i, B_i), \omega_i \rangle$ for every $i \geq 0$.

Definition 5.2. *A Symbolic Büchi Pushdown System (SBPDS) is a tuple (P, Γ, Δ, F), where (P, Γ, Δ) is a SPDS and $F \in P$ is a set of accepting control locations. A run of a SBPDS is accepting iff it visits infinitely often some control locations in F.*

Definition 5.3. *A Generalized Symbolic Büchi Pushdown System (GSBPDS) 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 a GSBPDS is accepting iff it visits infinitely often some control locations in F_i for every $1 \leq i \leq k$.*

Let $\mathcal{BP}$ be a SBPDS (resp. GSBPDS), $\mathcal{L}(\mathcal{BP})$ is the set of configurations $\langle (p, B), \omega \rangle \in P \times \mathcal{B} \times \Gamma^*$ such that $\mathcal{BP}$ has an accepting run from $\langle p, \omega \rangle$. We have the following properties:

Proposition 5.1. *[15] Given a GSBPDS $\mathcal{BP}$, we can compute a SBPDS $\mathcal{BP}'$ s.t. $\mathcal{L}(\mathcal{BP}) = \mathcal{L}(\mathcal{BP}')$.*

Theorem 5.1. *[5, 15] Given a SBPDS $\mathcal{P} = (P, \Gamma, \Delta, F)$, 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 $\mathcal{L}(\mathcal{BP})$ can be decided in time $O(|P|.|\Delta|^2.|\mathcal{D}|^{3|\mathcal{X}|})$.*

5.2 From PCARET$^{\backslash c}$ Model-Checking for PDSs to the Emptiness Problem of SBPDSs

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, $\lambda : P \to 2^{AP_{\mathcal{D}}}$ be a labelling function, ψ be a PCARET$^{\backslash c}$ formula. In this section, we show how to build a Generalized Symbolic Büchi Pushdown System $\mathcal{BP}_\psi$ s.t. $\mathcal{P}$ has an execution π from $\langle p, \omega\rangle$ s.t. π satisfies ψ under B iff $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \{\psi\}|\!), B), \omega\rangle$. Let $cl_{U^g}(\psi) = \{\phi_1 U^g \phi_1, ..., \phi_k U^g \phi_k\}$ and $cl_{U^a}(\psi) = \{\varphi_1 U^a \varphi_1, ..., \varphi_{k'} U^g \varphi_{k'}\}$ be the set of U^g-formulas and U^a-formulas of $Cl(\psi)$ respectively.

We define $\mathcal{BP}_\psi = (P', \Gamma', \Delta', F)$ as follows:

- $P' = P \times 2^{Cl(\psi)}$
- $\Gamma' = \Gamma \cup (\Gamma \times 2^{Cl(\psi)})$ is the finite set of stack symbols of $\mathcal{BP}_\psi$.
- $F = \{P \times F_{\phi_1 U^g \chi_1}, ..., P \times F_{\phi_k U^g \chi_k}, P \times F_{\xi_1 U^a \tau_1} ... P \times F_{\xi_{k'} U^a \tau_{k'}}\}$ where $F_{\phi_i U^g \chi_i} = \{\Phi \subseteq Cl(\psi) \mid$ if $\phi_i U^g \chi_i \in \Phi$ then $\chi_i \in \Phi\}$ for every $1 \le i \le k$; $F_{\xi_i U^a \tau_i} = \{\Phi \subseteq Cl(\psi) \mid$ if $\xi_i U^a \tau_i \in \Phi$ then $\tau_i \in \Phi\}$ for every $1 \le i \le k'$.

Δ' is the smallest set of transition rules defined as follows: for every $\Phi \subseteq Cl(\psi), p \in P, \gamma \in \Gamma$, [1]:

(β_1) if $\phi = b(\alpha_1, .., \alpha_n) \in \Phi$, then, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta} \langle(\!|p, \Phi \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$
where $\theta = \{(B, B) \mid B \in \mathcal{B} \wedge b((B(\alpha_1), ..., B((\alpha_n))) \in \lambda(p)\}$

(β_2) if $\phi = \neg b(\alpha_1, .., \alpha_n) \in \Phi$, then, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta} \langle(\!|p, \Phi \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$
where $\theta = \{(B, B) \mid B \in \mathcal{B} \wedge b((B(\alpha_1), ..., B((\alpha_n))) \notin \lambda(p)\}$

(β_3) if $\phi = \phi_1 \wedge \phi_2 \in \Phi$, then, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|p, \Phi \setminus \{\phi\} \cup \{\phi_1, \phi_2\}|\!), \gamma\rangle \in \Delta'$

(β_4) if $\phi = \phi_1 \vee \phi_2 \in \Phi$, then, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|p, \Phi \setminus \{\phi\} \cup \{\phi_1\}|\!), \gamma\rangle \in \Delta'$ and $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|p, \Phi \setminus \{\phi\} \cup \{\phi_2\}|\!), \gamma\rangle \in \Delta'$

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

($\beta_{5.1}$) if x is not a free variable of any formula in Φ, then, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_x} (\!|p, \Phi \cup \{\phi'\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$

($\beta_{5.2}$) otherwise, for every $c \in \mathcal{D}$, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} (\!|p, \Phi \cup \{\phi'_c\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ where ϕ'_c is ϕ' where x is substituted by c.

(β_6) if $\phi = \forall x \phi' \in \Phi$, then, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|p, \Phi \cup \{\phi'_c \mid c \in \mathcal{D}\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ where ϕ'_c is ϕ' such that x is replaced by c.

(β_7) if $\phi = \phi_1 U^v \phi_2 \in \Phi$ ($v \in \{g, a\}$), then, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|p, \Phi \cup \{\phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ and $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|p, \Phi \cup \{\phi_1, X^v \phi\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$

(β_8) if $\phi = \phi_1 R^v \phi_2 \in \Phi$ ($v \in \{g, a\}$), then, $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$ and $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|p, \Phi \cup \{\phi_2, X^v \phi\} \setminus \{\phi\}|\!), \gamma\rangle \in \Delta'$

(β_9) if $\Phi = \Phi_g \cup \Phi_a$ where $\Phi_g = \{X^g \phi_1, ..., X^g \phi_n\}$, $\Phi_a = \{X^a \varphi_1, ..., X^a \varphi_m\}$ (Φ_g or Φ_a can be empty), then:

($\beta_{9.1}$) for every $\langle p, \gamma\rangle \xrightarrow{call} \langle q, \gamma'\gamma''\rangle \in \Delta$:
$\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|q, \{\phi_1, ..., \phi_n\}|\!), \gamma'(\!|\gamma'', \{\varphi_1, ..., \varphi_m\}|\!)\rangle \in \Delta'$

($\beta_{9.2}$) for every $\langle p, \gamma\rangle \xrightarrow{ret} \langle q, \epsilon\rangle \in \Delta$:

($\beta_{9.2.1}$) $\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|q, \{\phi_1, ..., \phi_n\}|\!), \epsilon\rangle \in \Delta'$

($\beta_{9.2.2}$) $\langle(\!|q, \{\phi_1, ..., \phi_n\}|\!), (\!|\gamma_0, \Phi_0|\!)\rangle \xrightarrow{\theta_{id}} \langle(\!|q, \{\phi_1, ..., \phi_n\}|\!), \gamma_0\rangle \in \Delta'$ for every $\gamma_0 \in \Gamma$, $\Phi_0 \subseteq \{\phi_1, ..., \phi_n\}$

($\beta_{9.3}$) for every $\langle p, \gamma\rangle \xrightarrow{int} \langle q, \omega\rangle \in \Delta$:
$\langle(\!|p, \Phi|\!), \gamma\rangle \xrightarrow{\theta_{id}} \langle(\!|q, \{\phi_1, ..., \phi_n, \varphi_1, ..., \varphi_m\}|\!), \omega\rangle \in \Delta'$

Roughly speaking, we construct $\mathcal{BP}_\psi$ as a kind of product between $\mathcal{P}$ and ψ which ensures that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \{\psi\}|\!), B), \omega\rangle$ iff $\mathcal{P}$ has an execution π starting at $\langle p, \omega\rangle$ s.t. π satisfies ψ under B. The form of control locations of $\mathcal{BP}_\psi$ is $(\!|p, \Phi|\!)$ where Φ is a set of formulas. Let $\pi = \langle p_0, \omega_0\rangle\langle p_1, \omega_1\rangle...$ be a run of $\mathcal{P}$. Let us write $\pi \models^B_\lambda \Phi$ to express that π satisfies all formulas $\phi \in \Phi$ under B. To obtain such a $\mathcal{BP}_\psi$, intuitively, we proceed as follows:

- If $b(\alpha_1, .., \alpha_n) \in \Phi$, then $\pi \models^B_\lambda \Phi$ iff π satisfies $b(\alpha_1, .., \alpha_n)$ and π satisfies all the remaining formulas in Φ under B. This is ensured by the transition rules in (β_1) stating that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi|\!), B), \omega\rangle$ iff $b(B(\alpha_1), .., B(\alpha_n)) \in \lambda(p)$ and $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi \setminus \{b(\alpha_1, .., \alpha_n)\}|\!), B), \omega\rangle$.
- If $\neg b(\alpha_1, .., \alpha_n) \in \Phi$, then $\pi \models^B_\lambda \Phi$ iff π satisfies $\neg b(\alpha_1, .., \alpha_n)$ and π satisfies all the remaining formulas in Φ under B. This is ensured by the transition rules in (β_2) stating that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi|\!), B), \omega\rangle$ iff $b(B(\alpha_1), .., B(\alpha_n)) \notin \lambda(p)$ and $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p, \Phi \setminus \{\neg b(\alpha_1, .., \alpha_n)\}|\!), B), \omega\rangle$.
- If $\phi_1 \wedge \phi_2 \in \Phi$, then $\pi \models^B_\lambda \Phi$ iff π satisfies $\phi_1 \wedge \phi_2$ and π satisfies all the remaining formulas in Φ under B. Note that $\pi \models^B_\lambda \phi_1 \wedge \phi_2$ iff $\pi \models^B_\lambda \phi_1$ and $\pi \models^B_\lambda \phi_2$. This is ensured by the transition rules in (β_3) 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 $\langle((\!|p, \Phi \cup \{\phi_1, \phi_2\} \setminus \{\phi_1 \wedge \phi_2\}|\!), B), \omega\rangle$. Item ($\beta_4$) is similar to ($\beta_3$)
- If $\forall x \phi' \in \Phi$, then $\pi \models^B_\lambda \Phi$ iff π satisfies $\forall x \phi'$ and π satisfies all the remaining formulas in Φ under B. Note that $\pi \models^B_\lambda \forall x \phi'$ iff $\pi \models^B_\lambda \bigwedge_{c \in \mathcal{D}} \phi'_c$ where ϕ'_c is ϕ' in which x is replaced by c. Thus, this is ensured by the transition rules in (β_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 $\langle((\!|p, \Phi \cup \{\phi'_c | c \in \mathcal{D}\} \setminus \{\forall x \phi'\}|\!), B), \omega\rangle$.
- If $\exists x \phi' \in \Phi$, then, $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda \exists x \phi'$ and $\pi \models^B_\lambda \Phi \setminus \{\exists x \phi'\}$. In other words, $\pi \models^B_\lambda \Phi$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models^{B[x \leftarrow c]}_\lambda \phi'$ and $\pi \models^B_\lambda \Phi \setminus \{\exists x \phi'\}$. We consider two possibilities:

[1] θ_x and θ_{id} are defined in Section 3.1

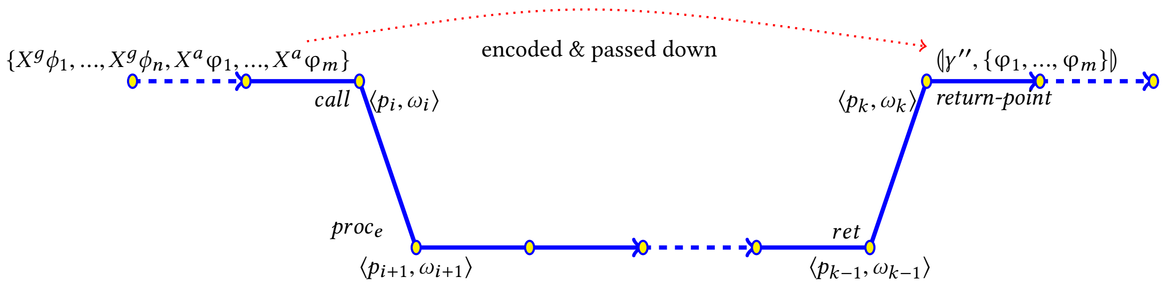
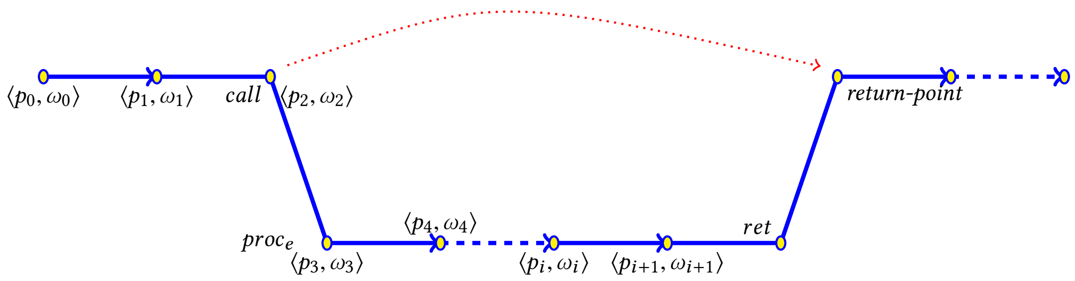

$\{X^g\phi_1,...,X^g\phi_n,X^a\varphi_1,...,X^a\varphi_m\}$
encoded & passed down
$(\!|\gamma'',\{\varphi_1,...,\varphi_m\}|\!)$
$call$
$\langle p_i,\omega_i\rangle$
$\langle p_k,\omega_k\rangle$
$return\text{-}point$
$proc_e$
ret
$\langle p_{i+1},\omega_{i+1}\rangle$
$\langle p_{k-1},\omega_{k-1}\rangle$

Figure 2: $\langle p_i,\omega_i\rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1},\omega_{i+1}\rangle$ corresponds to a call statement

- if x is not a free variable of any formula in Φ, then, $\pi \models^B_\lambda \Phi \setminus \{\exists x\phi'\}$ iff $\pi \models^{B[x\leftarrow c]}_\lambda \Phi \setminus \{\exists x\phi'\}$ for every $c \in \mathcal{D}$. This means 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 the transition rules in $(\beta_{5.1})$ 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 $\langle((\!|p,\Phi \cup \{\phi'\} \setminus \{\exists x\phi'\}|\!),B[x \leftarrow c]),\omega\rangle$ (as $(B,B[x \leftarrow c]) \in \theta_x$)
- if x is a free variable of some formula in Φ, then, it may occur the case that ϕ' is satisfied only when $x = c$ ($\pi \models^{B[x\leftarrow c]}_\lambda \phi'$), ϕ'' is not satisfied when $x = c$ ($\pi \not\models^{B[x\leftarrow c]}_\lambda \phi''$), while $\pi \models^B_\lambda \{\exists x\phi',\phi''\}$. Thus, we cannot apply the transition rules in $(\beta_{5.1})$ for this case. Note that $\pi \models^B_\lambda \Phi$ iff there exists $c \in \mathcal{D}$ s.t. $\pi \models^{B[x\leftarrow c]}_\lambda \phi'$ and $\pi \models^B_\lambda \Phi \setminus \{\exists x\phi'\}$. This is ensured by the transition rule $(\beta_{5.2})$ 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 $\langle((\!|p,\Phi \cup \{\phi'_c\} \setminus \{\exists x\phi'\}|\!),B),\omega\rangle$ (since $(B,B) \in \theta_{id}$).

- If $\phi_1 U^v \phi_2 \in \Phi$, then $\pi \models^B_\lambda \Phi$ iff $\pi \models^B_\lambda \phi_1 U^v \phi_2$ and $\pi \models^B_\lambda \{\Phi \setminus \phi_1 U^v \phi_2\}$. Note that $\pi \models^B_\lambda \phi_1 U^v \phi_2$ iff $\pi \models^B_\lambda \phi_2$ or ($\pi \models^B_\lambda \phi_1$ and $\pi \models^B_\lambda X^v(\phi_1 U^v \phi_2)$). This is ensured by the transition rules in (β_7) stating that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p,\Phi|\!),B),\omega\rangle$ iff either $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p,\Phi \cup \{\phi_2\} \setminus \{\phi_1 U^v \phi_2\}|\!),B),\omega\rangle$, or $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p,\Phi \cup \{\phi_1, X^v(\phi_1 U^v \phi_2)\} \setminus \{\phi_1 U^v \phi_2\}|\!),B),\omega\rangle$. Item (β_8) is similar to (β_7).
- If $\Phi = \{X^g\phi_1,...,X^g\phi_n,X^a\varphi_1,...,X^a\varphi_m\}$. Let $\langle p_k,\omega_k\rangle$ be the abstract-successor of $\langle p_i,\omega_i\rangle$, then, $(\pi,i) \models^B_\lambda \Phi$ iff $((\pi,i+1) \models^B_\lambda \{\phi_1,...,\phi_n\}$ and $(\pi,k) \models^B_\lambda \{\varphi_1,...,\varphi_m\})$. Now we show how we can ensure these:
  - $(\pi,i+1) \models^B_\lambda \{\phi_1,...,\phi_m\}$ is ensured by the transition rules corresponding to different cases in $(\beta_{9.1})$, $(\beta_{9.2})$ and $(\beta_{9.3})$ which guarantee that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p,\{X^g\phi_1,...,X^g\phi_n,X^a\varphi_1,...,X^a\varphi_m\}|\!),B),\omega\rangle$ iff $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|q,\{\phi_1,...,\phi_m\}|\!),B),\omega'\rangle$.
  - To ensure $(\pi,k) \models^B_\lambda \{\varphi_1,...,\varphi_m\}$ There are two possibilities:
    * If $\langle p_i,\omega_i\rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1},\omega_{i+1}\rangle$ corresponds to a call statement. Let us consider Figure 2 to explain this case. $(\pi,i) \models^B_\lambda \{X^a\varphi_1,...,X^a\varphi_m\}$ if $(\pi,k) \models^B_\lambda \{\varphi_1,...,\varphi_m\}$. This is ensured by rules $(\beta_{9.1})$ and $(\beta_{9.2})$: rules $(\beta_{9.1})$ allow to record $\{\varphi_1,...,\varphi_m\}$ in the return point of the call, and rules $(\beta_{9.2})$ allow to extract and validate $\{\varphi_1,...,\varphi_m\}$ when the return-point is reached. In what follows, we show in more details how this works: Let $\langle p_i,\gamma\rangle \xrightarrow{call} \langle p_{i+1},\gamma'\gamma''\rangle$ be the rule associated with the transition $\langle p_i,\omega_i\rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1},\omega_{i+1}\rangle$, then we have $\omega_i = \gamma\omega'$ and $\omega_{i+1} = \gamma'\gamma''\omega'$. Let $\langle p_{k-1},\omega_{k-1}\rangle \Rightarrow_{\mathcal{P}} \langle p_k,\omega_k\rangle$ be the transition that corresponds to the *ret* statement of this call.
    Let then $\langle p_{k-1},\beta\rangle \xrightarrow{ret} \langle p_k,\epsilon\rangle \in \Delta$ be the corresponding return rule. Then, we have necessarily $\omega_{k-1} = \beta\gamma''\omega'$, since as explained in Section 2.1, γ'' is the return address of the call. After applying this rule, $\omega_k = \gamma''\omega'$. In other words, γ'' will be the topmost stack symbol at the corresponding return point of the call. So, in order to ensure that $(\pi,k) \models^B_\lambda \{\varphi_1,...,\varphi_m\}$, we proceed as follows: At the call $\langle p_i,\gamma\rangle \xrightarrow{call} \langle p_{i+1},\gamma'\gamma''\rangle$, we encode formulas which are required to be true at the corresponding return-point of the call $\{\varphi_1,...,\varphi_m\}$ into γ'' by the rule $(\beta_{9.1})$ stating that $\langle(\!|p_i,\{X^g\phi_1,...,X^g\phi_n,X^a\varphi_1,...,X^a\varphi_m\}|\!),\gamma\rangle \longrightarrow \langle(\!|p_{i+1},\{\phi_1,...,\phi_n\}|\!),\gamma'(\!|\gamma'',\{\varphi_1,...,\varphi_m\}|\!)\rangle \in \Delta'$. This allows to record $\{\varphi_1,...,\varphi_m\}$ in the corresponding return point of the stack. After that, $(\!|\gamma'',\{\varphi_1,...,\varphi_m\}|\!)$ will be the topmost stack symbol at the corresponding return-point of this call. At the return-point, the condition in $(\beta_{9.2.2})$ ensures that $(\pi,k) \models^B_\lambda \{\varphi_1,...,\varphi_m\}$.
    * If $\langle p_i,\omega_i\rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1},\omega_{i+1}\rangle$ corresponds to a simple statement (see Figure 3). Then, the abstract successor of $\langle p_i,\omega_i\rangle$ is $\langle p_{i+1},\omega_{i+1}\rangle$. Thus, we must have $(\pi,i+1) \models^B_\lambda \{\varphi_1,...,\varphi_m\}$. This is ensured by the rules in $(\beta_{9.3})$ stating that $\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p_i,\Phi_i|\!),B_i),\omega_i\rangle$ iff



$\langle p_0, \omega_0\rangle$
$\langle p_1, \omega_1\rangle$
call
$\langle p_2, \omega_2\rangle$
return-point
$\langle p_4, \omega_4\rangle$
$proc_e$
$\langle p_3, \omega_3\rangle$
ret
$\langle p_i, \omega_i\rangle$
$\langle p_{i+1}, \omega_{i+1}\rangle$

Figure 3: $\langle p_i, \omega_i\rangle \Rightarrow_{\mathcal{P}} \langle p_{i+1}, \omega_{i+1}\rangle$ corresponds to a simple statement

$\mathcal{BP}_\psi$ has an accepting run from $\langle((\!|p_{i+1}, \{\phi_1, ..., \phi_n, \varphi_1, ..., \varphi_m\}|\!), B_{i+1}), \omega_{i+1}\rangle$.

The generalized Büchi accepting condition F of $\mathcal{BP}_\psi$ is determined by a set of sets where each set of F ensures that the liveness requirement ϕ_2 in $\phi_1 U^v \phi_2$ ($v \in \{g, a\}$) is eventually satisfied in $\mathcal{P}$. Note that $(\pi, i) \models^B_\lambda \phi_1 U^v \phi_2$ iff $(\pi, i) \models^B_\lambda \phi_2$ or $((\pi, i) \models^B_\lambda \phi_1$ and $(\pi, i) \models^B_\lambda X^v(\phi_1 U^v \phi_2))$. Because ϕ_2 should hold eventually, to avoid the case where the run of $\mathcal{BP}_\psi$ always carries (ϕ_1 and $X^v(\phi_1 U^v \phi_2)$) and never reaches ϕ_2, we set $P \times F_{\phi_1 U^v \phi_2} = P \times \{\Phi \subseteq Cl(\psi) \mid$ if $\phi_1 U^v \phi_2 \in \Phi$ then $\phi_2 \in \Phi\}$ as a set of Büchi generalized accepting condition. By this setting, the accepting run of $\mathcal{BP}_\psi$ will infinitely often visit some control locations in $P \times \{\Phi \subseteq Cl(\psi) \mid$ if $\phi_1 U^v \phi_2 \in \Phi$ then $\phi_2 \in \Phi\}$ which ensures that ϕ_2 will eventually hold.

Thus, we can show that:

Theorem 5.2. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : P \longrightarrow 2^{AP_\mathcal{D}}$, and a PCARET$^{\backslash c}$ formula ψ, we can construct a GSBPDS $\mathcal{BP}_\psi = (P', \Gamma', \Delta', F)$ such that for every configuration $\langle p, \omega\rangle \in P \times \Gamma^*$ and every $B \in \mathcal{B}$, $\langle p, \omega\rangle$ satisfies ψ under B iff $\langle((\!|p, \{\psi\}|\!), B), \omega\rangle \in \mathcal{L}(\mathcal{BP}_\psi)$.*

Remark 5.1. *Note that the above procedure could not be applied if we consider the operators X^c, U^c and R^c. Indeed, the key point of our construction is that we use the symbolic transitions of SBPDS to express different possible values of variables which allow us to obtain a better complexity. To do that, our construction is based on the fact that the satisfiability of a given formula at a certain state is ensured by the satisfiability of several formulas at the successor state. However, if we allow X^c, this does not hold anymore. Let us consider Figure 3 to illustrate this. For instance, we want to determine whether $\langle p_0, \omega_0\rangle \models^B_\lambda X^g X^g X^g X^g X^c \phi$. This holds iff $\langle p_1, \omega_1\rangle \models^B_\lambda X^g X^g X^g X^c \phi$ (this is expressed by rules ($\beta_{9.3}$)), iff $\langle p_2, \omega_2\rangle \models^B_\lambda X^g X^g X^c \phi$, iff $\langle p_3, \omega_3\rangle \models^B_\lambda X^g X^c \phi$, iff $\langle p_4, \omega_4\rangle \models^B_\lambda X^c \phi$, iff $\langle p_2, \omega_2\rangle \models^B_\lambda \phi$ .The four first requirements are ensured by rules ($\beta_{9.3}$)), since they consider the immediate successor state. However, the last requirement cannot be ensured by a rule like the (β) rules above since the caller successor of a state is a predecessor of that state.*

Remark 5.2. *One can wonder why we do not apply the approach proposed in [11] which uses atoms (maximally consistent subsets of $Cl(\psi)$) to deal with all CARET operators (including X^c, U^c and R^c). The reason is that SPCARET contains variables and quantifiers over variables, which makes it impossible to compute atoms without enumerating all possible values of variables. We could do this, but this has exactly the same complexity as translating the SPCARET formula into CARET and then model checking the CARET formula, since it is based on enumerating all possible values. Thus, we cannot benefit from the symbolic representation of variables using this approach.*

Remark 5.3. *Note that the transition rules in ($\beta_{5.2}$) are never applied if there are no free variables in the formula ψ. The key point of our construction lies in the case $\exists x \phi' \in \Phi$. Indeed, for this case, our construction have moved the complexity from the formula to the transitions of the* symbolic *Büchi pushdown system, where all possible values of the variable x are symbolically encoded within the symbolic relation θ_x. Let us take a simple example to illustrate this case. Consider this simple PCARET$^{\backslash c}$ formula $\psi = \exists x \exists y\ mov(x, y)$. To model check this formula against Pushdown Systems, we have two approaches:*

- *Translate ψ to an equivalent CARET formula and apply the algorithm presented in [11]. We obtain the equivalent CARET formula $\psi' = \bigvee_{x \in D} \bigvee_{y \in D} mov(x, y)$. Note that $|\psi'| = |\psi| \times O(|D|^{|X|})$. Roughly speaking, we will consider all possible combinations of values of the tuple of variables (x, y) which is very large.*
- *Apply our above algorithm. For the case $\psi = \exists x \exists y\ mov(x, y)$, by applying the transition rules in ($\beta_{5.1}$), the different values of x are represented by* one *symbolic relation θ_x, and the different values of y are represented by* one *symbolic relation θ_y. These relations can be efficiently represented using BDDs as explained in [5]. Thus, our algorithms works very well for this case.*

The complexity of the above algorithm depends on whether the formula contains universal quantifiers or not:

- If ψ contains only existential quantifiers and no free variables, then, the number of control locations of $\mathcal{BP}_\psi$ is at most $|P| \times 2^{O(|\psi|)}$ and the number of transitions is at most $|\Delta||\Gamma| \times 2^{O(|\psi|)}$. From Theorem 5.1, the membership problem can be solved in time $|P|.|\Delta|^2.|\Gamma|^2.2^{O(|\psi|)}|\mathcal{D}|^{3|X|}$.
- If ψ contains universal quantifiers or free variables, for the worst case, the number of control locations of $\mathcal{BP}_\psi$ is at most $|P| \times 2^{O(|\psi||\mathcal{D}|^{|X|})}$ and the number of transitions is at

most $|\Delta||\Gamma|\times 2^{O(|\psi||\mathcal{D}|^{|\mathcal{X}|})}$. From Theorem 5.1, the membership problem can be solved in time $|P|.|\Delta|^2.|\Gamma|^2.2^{O(|\psi||\mathcal{D}|^{|\mathcal{X}|})}|\mathcal{D}|^{3|\mathcal{X}|}$.

Thus, we get:

Theorem 5.3. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : P \longrightarrow 2^{AP_{\mathcal{D}}}$ and a PCARET$^{\backslash c}$ formula ψ, for every configuration $\langle p, \omega\rangle$ and every $B \in \mathcal{B}$, whether or not $\langle p, \omega\rangle$ satisfies ψ under B can be solved in time $|P|.|\Delta|^2.|\Gamma|^2.2^{O(|\psi||\mathcal{D}|^{|\mathcal{X}|})}|\mathcal{D}|^{3|\mathcal{X}|}$. Moreover, if ψ contains only existential quantifiers, and no free variables, then this problem can be solved in time $|P|.|\Delta|^2.|\Gamma|^2.2^{O(|\psi|)}|\mathcal{D}|^{3|\mathcal{X}|}$.*

Remark 5.4. *Formulas that describe malicious behaviors do not involve free variables and contain only existential quantifiers since the malware detection problem consists in determining whether there exists a path or a value of a variable for which the malicious behaviour occurs. Thus, for malware detection, our algorithm has a better complexity than translating the PCARET formula into a CARET formula, and then apply the CARET model checking algorithm.*

6 SPCARET$^{\backslash c}$ MODEL-CHECKING FOR PDS

In this section, we discuss how to do SPCARET$^{\backslash c}$ model-checking for PDSs. Let then $\mathcal{P}$ be a PDS, ψ be a SPCARET$^{\backslash c}$ formula, and $\mathcal{V}$ be the set of RVEs occuring in ψ. We follow the idea of [15] and use Extended Finite Automata to represent RVEs.

Definition 6.1. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, let $\varpi = \{\alpha, \neg\alpha \mid \alpha \in \Gamma\cup\mathcal{X}\}$, an Extended Finite Automaton (EFA) $\mathcal{K}$ is a tuple $(Q, \Lambda, \Gamma, q_0, Q_f)$ s.t Q is a finite set of states, q_0 is the initial state, $Q_f \subseteq Q$ is a finite set of final states, Γ is a finite set of alphabet, Λ is a finite of transitions rules in the form $q_1 \xrightarrow{l} q_2$ where $q_1, q_2 \in Q, l \in \varpi$.*

Let $B \in \mathcal{B}$ be an environment, $\gamma \in \Gamma$ be the input letter, $t = q_1 \xrightarrow{l} q_2$ be a transition rule in Λ, assume that $\mathcal{K}$ is at state q_1, then, $\mathcal{K}$ can move to the state q_2 under B (denoted $q_1 \xrightarrow{\gamma}_B q_2$) iff for every $\alpha \in l$, $B(\alpha) = \gamma$ and for every $\neg\alpha \in l$, $B(\alpha) \neq \gamma$. $\mathcal{K}$ accepts a word $\gamma_0 ... \gamma_n$ under B iff $\mathcal{K}$ has a run $q_0 \xrightarrow{\gamma_0}_B q_1 ... q_n \xrightarrow{\gamma_n}_B q_{n+1}$ where $q_{n+1} \in Q_f$. Let $L(\mathcal{K})$ be the set of all configurations $(\langle p, \omega\rangle, B) \in P \times \Gamma^* \times \mathcal{B}$ s.t $\mathcal{K}$ accepts ω under B.

Proposition 6.1. *[15] For every RVE $e \in \mathcal{V}$, we can compute in polynomial time an EFA $\mathcal{K}_e$ s.t $L(e) = L(\mathcal{K}_e)$.*

To do SPCARET$^{\backslash c}$ model-checking for PDSs, we first need to represent each RVE $e \in \mathcal{V}$ by an EFA $\mathcal{K}$ s.t $\mathcal{K}$ recognizes all the configurations $(\langle p, \omega\rangle, B) \in P \times \Gamma^* \times \mathcal{B}$ in $L(e)$. Let $\mathcal{K}^1, ..., \mathcal{K}^n$ be the set of automata corresponding to all the RVEs of $\mathcal{V}$. Then, we compute a Symbolic Pushdown System (SPDS) $\mathcal{P}_0$ which is a kind of product between $\mathcal{P}$ and $\mathcal{K}^1, ..., \mathcal{K}^n$ which allows us to determine whether the stack predicates hold or not only by looking at the top of the stack of $\mathcal{P}_0$. Then, SPCARET model checking for $\mathcal{P}$ is reduced to PCARET model-checking for $\mathcal{P}_0$. We adapt then the algorithms in Section 5.2 to deal with Symbolic PDSs. Due to lack of space, the adaptation is given in the full version of the paper [10]. Thus, we get that:

Theorem 6.1. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a labeling function $\lambda : P \longrightarrow 2^{AP_{\mathcal{D}}}$, and a SPCARET$^{\backslash c}$ formula ψ, we can construct a GSBPDS $\mathcal{BP}_\psi = (P', \Gamma', \Delta', F)$ such that for every configuration $\langle p, \omega\rangle \in P \times \Gamma^*$ and every $B \in \mathcal{B}$, $\langle p, \omega\rangle$ satisfies ψ under B iff $\langle((\!|p, \{\psi\}|\!), B), \omega'\rangle \in \mathcal{L}(\mathcal{BP}_\psi)$ where ω' is obtained by performing the product between ω and the EFAs $\mathcal{K}^1, ..., \mathcal{K}^n$.*

7 EXPERIMENTS

We implemented our algorithms in a tool for malware detection. We use IDA Pro [6], Jakstab [8] and the translation of [13] to obtain PDSs from binary code or assembly programs. We use Moped [5] to check the emptiness of SBPDSs. A program is considered to be a malware if it satisfies one of the SPCARET formulas presented previously, otherwise, it is a benign program. Our tool was able to detect several malwares and to determine that benign programs are benign as reported in Table 1. The **#LOC** column shows the number of instructions of the assembly program. The result *Yes* expresses that the binary program is detected as a malware, *No* means the program is found as benign.

Moreover, we compared the performance of our algorithms against the approach that consists in translating SPCARET into CARET with regular valuations, and applying the model checking algorithm for CARET. We set the time limit to 20 minutes. You can see in Table 1 that we perform much better in all cases, and that in most cases, the approach that consists in translating SPCARET to CARET timeout.

	Samples	#LOC	SPCARET		CARET	
			Times(s)	Result	Times (s)	Result
Malware	Trojan-Downloader.Win32.Apropo.al	17870	41.25	Yes	timeout	
	Trojan-Downloader.Win32.Apropo.bb	17785	32.36	Yes	timeout	
	Virus.Win32.NGVCK.1095	3137	54.17	Yes	timeout	
	Virus.Win32.Redemption.b	1486	23.05	Yes	timeout	
	Email-Worm.Win32.Lohack.a	4887	24.84	Yes	timeout	
	Email-Worm.Win32.Scaline.a	8207	89.99	Yes	timeout	
	IRC-Worm.Win32.Azrael	3302	58.74	Yes	timeout	
	Net-Worm.Win32.Nimda	8670	60.08	Yes	timeout	
	Trojan-Downloader.Win32.Apropo.bd	27071	53.63	Yes	timeout	
	Trojan-Downloader.Win32.Apropo.s	17902	37.13	Yes	timeout	
	Trojan-Downloader.Win32.Apropo.x	45360	68.89	Yes	timeout	
	Trojan-Downloader.Win32.Dyfuca.ab	11041	113.09	Yes	timeout	
	Email-Worm.Win32.NetSky.a	6352	90.27	Yes	874.24	Yes
	Email-Worm.Win32.Klez.e	14562	70.41	Yes	timeout	
	Email-Worm.Win32.Klez.f	14570	75.82	Yes	timeout	
	Email-Worm.Win32.Klez.g	14582	82.61	Yes	timeout	
	Email-Worm.Win32.Klez.i	15042	77.16	Yes	timeout	
	Email-Worm.Win32.Bagle.al	764	32.51	Yes	timeout	
	Email-Worm.Win32.Bagle.m	5049	67.84	Yes	timeout	
	Trojan-PSW.Win32.LdPinch.aar	1197	27.45	Yes	timeout	
	Trojan-PSW.Win32.LdPinch.aoq	7592	43.82	Yes	timeout	
	Trojan-PSW.Win32.LdPinch.mj	5873	52.87	Yes	timeout	
Benign	cmd.exe	35856	112.43	No	timeout	
	ipv6.exe	12795	95.26	No	timeout	
	dplaysrv.exe	6820	41.52	No	timeout	
	regedt.exe	60	1.32	No	2.41	No
	blastcln.exe	13768	125.63	No	timeout	

Table 1: Detection of real malwares

REFERENCES

[1] Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS 2004*, 2004.

[2] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001.

[3] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Concurrency Theory, 1997*, 1997.

[4] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[5] Javier Esparza and Stefan Schwoon. A bdd-based model checker for recursive programs. In *CAV 2001*, 2001.

[6] Hex-Rays. IDAPRO. URL: https://www.hex-rays.com/products/ida/.

[7] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA 2005, Vienna, Austria, July 7-8, 2005, Proceedings*, pages 174–187, 2005.

[8] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *CAV*, 2008.

[9] Arun Lakhotia, Eric Uday Kumar, and Michael Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Trans. Software Eng.*, 31(11), 2005.

[10] Huu-Vu Nguyen and Tayssir Touili. CARET model checking for malware detection. URL: https://lipn.univ-paris13.fr/~nguyen/SPIN2017FullVersion.pdf.

[11] Huu-Vu Nguyen and Tayssir Touili. CARET model checking for pushdown systems. In *SAC 2017*, 2017.

[12] Stefan Schwoon. *Model-Checking Pushdown Systems*. Dissertation, Technische Universität München, München, 2002.

[13] Fu Song and Tayssir Touili. Efficient malware detection using model-checking. In *FM 2012: Formal Methods - 18th International Symposium*, 2012.

[14] Fu Song and Tayssir Touili. Pushdown model checking for malware detection. 2012.

[15] Fu Song and Tayssir Touili. LTL model-checking for malware detection. In *TACAS 2013*, 2013.

[16] Fu Song and Tayssir Touili. Pommade: pushdown model-checking for malware detection. In *SIGSOFT 2013*, 2013.