LTL Model Checking of Self Modifying Code.

Tayssir Touili, Xin Ye

LTL Model Checking of Self Modifying Code

Tayssir Touili
LIPN, CNRS and University Paris 13

Xin Ye
Shanghai Key Lab. of Trustworthy Comput., ECNU
and LIPN, CNRS and University Paris 13

***Abstract*—Self modifying code is code that can modify its own instructions during the execution of the program. It is extensively used by malware writers to obfuscate their malicious code. Thus, analysing self modifying code is nowadays a big challenge. In this paper, we consider the LTL model-checking problem of self modifying code. We model such programs using self-modifying pushdown systems (SM-PDS), an extension of pushdown systems that can modify its own set of transitions during execution. We reduce the LTL model-checking problem to the emptiness problem of self-modifying Büchi pushdown systems (SM-BPDS). We implemented our techniques in a tool that we successfully applied for the detection of several self-modifying malware. Our tool was also able to detect several malwares that well-known antiviruses such as BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Baidu, Avast, and Symantec failed to detect.**

I. Introduction

Binary code presents several complex aspects that cannot be encountred in source code. One of these aspects is self-modifying code, i.e., code that can modify its own instructions during the execution of the program. Self-modifying code makes reverse code engineering harder. Thus, it is extensively used to protect software intellectual property. It is also heavily used by malware writers in order to make their malwares hard to analyse and detect by static analysers and anti-viruses. Thus, it is crucial to be able to analyse self-modifying code.

There are several kinds of self-modifying code. In this work, we consider self-modifying code caused by **self-modifying instructions**. These kind of instructions treat code as data. This allows them to read and write into code, leading to **self-modifying instructions**. These self-modifying instructions are usually **mov** instructions, since **mov** allows to access memory and read and write into it.

Let us consider the example shown in Fig.1. For simplicity, the addresses' length is assumed to be 1 byte. In the right box, we give, respectively, the binary code, the addresses of the different instructions, and the corresponding assembly code, obtained by translating syntactically the binary code at each address. For example, `0c` is the binary code of the jump `jmp`. Thus, `0c 02` is translated to `jmp 0x2` (jump to address 0x2). The second line is translated to `push 0x9`, since `ff` is the binary code of the instruction `push`. The third instruction `mov 0x2 0xc` will replace the first byte at address `0x2` by `0xc`. Thus, at address `0x2`, `ff 09` is replaced by `0c 09`. This means the instruction `push 0x9` is replaced by the jump instruction `jmp 0x9` (jump to address 0x9), etc. Therefore, this code is self-modifying: the **mov** instruction was able to modify the instructions of the program via its ability to read and write the memory. If we study this code without looking at the semantics of the self-modifying instructions, we will extract from it the Control Flow Graph `CFG a` that is in the left of the figure, and we will reach the conclusion that the call to the API function CopyFileA at address `0x9` cannot be made. However, you can see that the correct CFG is the one on the right hand side `CFG b`, where the call to the API function CopyFileA at address `0x9` can be reached. Thus, it is very important to be able to take into account the semantics of the self-modifying instructions in binary code.

In this paper, we consider the LTL model-checking problem of self-modifying code. To this aim, we use Self-Modifying Pushdown Systems (SM-PDSs) [1] to model self-modifying code. Indeed, SM-PDSs were shown in [1] to be an adequate model for self-modifying code since they allow to mimic the program's stack while taking into account the self-modifying semantics of the transitions. This is very important for binary code analysis and malware detection, since malwares are based on calls to API functions of the operating system. Thus, antiviruses check the API calls to determine whether a program is malicious or not. Therefore, to evade from these antiviruses, malware writers try to hide the API calls they make by replacing calls by push and jump instructions. Thus, to be able to analyse such malwares, it is crucial to be able to analyse the program's stack. Hence the need to a model like pushdown systems and self-modifying pushdown systems for this purpose, since they allow to mimic the program's stack.

Intuitively, a SM-PDS is a pushdown system (PDS) with self-modifying rules, i.e., with rules that allow to modify the current set of transitions during execution. This model was introduced in [1] in order to represent self-modifying code. In [1], the authors have proposed algrithms to compute finite automata that accept the forward and backward reachability sets of SM-PDSs. In this work, we tackle the problem of LTL model-checking of SM-PDSs. Since SM-PDSs are equivalent to PDSs [1], one possible approach for LTL model checking of SM-PDS is to translate the SM-PDS to a standard PDS and then run the LTL model checking algorithm on the equivalent PDS [2], [3]. But translation from a SM-PDS to a standard PDS is exponential. Thus, performing the LTL model checking on the equivalent PDS is not efficient.

To overcome this limitation, we propose a *direct* LTL model checking algorithm for SM-PDSs. Our algorithm is based on reducing the LTL model checking problem to the emptiness problem of Self Modifying Büchi Pushdown Systems (SM-BPDS). Intuitively, we obtain this SM-BPDS by taking the product of the SM-PDS with a Büchi automaton accepting an LTL formula φ. Then, we solve the emptiness problem of an

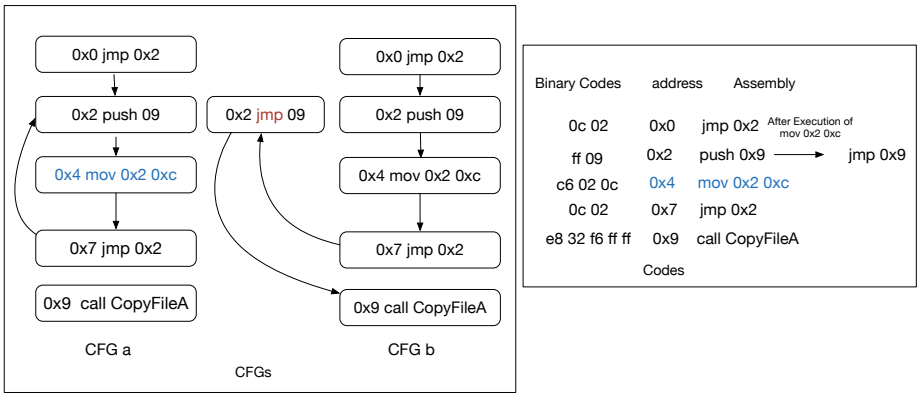

Fig. 1: An Example of a Self-modifying Code

SM-BPDS by computing its repeating heads. This computation is based on computing labelled pre^* configurations by applying a saturation procedure on labelled finite automata.

We implemented our algorithm in a tool. Our experiments show that our *direct* techniques are much more efficient than translating the SM-PDS to an equivalent PDS and then applying the standard LTL model checking for PDSs [2], [3]. Moreover, we successfully applied our tool to the analysis of 892 self-modifying malwares. Our tool was also able to detect several self-modifying malwares that well-known antiviruses like BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Baidu, Avast, and Symantec were not able to detect.

Related Work. Model checking and static analysis approaches have been widely used to analyze binary programs, for instance, in [4], [5], [6], [7], [8]. Temporal Logics were chosen to describe malicious behaviors in [9], [7], [8], [10], [11]. However, these works cannot deal with self-modifying code.

POMMADE [8], [10] is a malware detector based on LTL and CTL model-checking of PDSs. STAMAD [12], [13], [14] is a malware detector based on PDSs and machine learning. However, POMMADE and STAMAD cannot deal with self-modifying code.

Cai et al. [15] use local reasoning and separation logic to describe self-modifying code and treat program code uniformly as regular data structure. However, [15] requires programs to be manually annotated with invariants. In [16], the authors propose a formal semantics for self-modifying codes, and use that to represent self-unpacking code. This work only deals with packing and unpacking behaviours. Bonfante et al. [17] provide an operational semantics for self-modifying programs and show that they can be constructively rewritten to a non-modifying program. However, all these specifications [17], [15], [16] are too abstract to be used in practice.

In [18], the authors propose a new representation of self-modifying code named State Enhanced-Control Flow Graph (SE-CFG). SE-CFG extends standard control flow graphs with a new data structure, keeping track of the possible states programs can reach, and with edges that can be conditional on the state of the target memory location. It is not easy to analyse a binary program only using its SE-CFG, especially that this representation does not allow to take into account the stack of the program.

[19] propose abstract interpretation techniques to compute an over-approximation of the set of reachable states of a self-modifying program, where for each control point of the program, an over-approximation of the memory state at this control point is provided. [20] combine static and dynamic analysis techniques to analyse self-modifying programs. Unlike our approach, these techniques [19], [20] cannot handle the program's stack.

Unpacking binary code is also considered in [21], [22], [23], [16]. These works do not consider self-modifying **mov** instructions.

Outline. The rest of the paper is structured as follows: Section 2 recalls the definition of Self Modifying pushdown systems. LTL model checking and SM-BPDSs are defined in Section 3. Section 4 solves the emptiness problem of SM-BPDS. Finally, the experiments are reported in Section 5.

II. Self Modifying Pushdown Systems

A. Definition

We recall in this section the definition of Self-modifying Pushdown Systems [1].

Definition 1. *A Self-modifying Pushdown System (SM-PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$, where P is a finite set of control points, Γ is a finite set of stack symbols, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules, and $\Delta_c \in P \times \Delta \times \Delta \times P$ is a finite set of modifying transition rules. If $((p, \gamma), (p', w)) \in \Delta$, we also write $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$. If $(p, r_1, r_2, p') \in \Delta_c$, we also write $p \xrightarrow{(r_1, r_2)} p' \in \Delta_c$. A Pushdown System (PDS) is a SM-PDS where $\Delta_c = \emptyset$.*

Intuitively, a Self-modifying Pushdown System is a Pushdown System that can dynamically modify its set of rules during the execution time: rules Δ are standard PDS transition rules, while rules Δ_c modify the current set of transition rules: $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$ expresses that if the SM-PDS is in control point p and has γ on top of its stack, then it can move

to control point p', pop γ and push w onto the stack, while $p \xhookrightarrow{(r_1,r_2)} p' \in \Delta_c$ expresses that when the PDS is in control point p, then it can move to control point p', remove the rule r_1 from its current set of transition rules, and add the rule r_2.

Formally, a configuration of a SM-PDS is a tuple $c = (\langle p, w\rangle, \theta)$ where $p \in P$ is the control point, $w \in \Gamma^*$ is the stack content, and $\theta \subseteq \Delta \cup \Delta_c$ is the current set of transition rules of the SM-PDS. θ is called the current *phase* of the SM-PDS. When the SM-PDS is a PDS, i.e., when $\Delta_c = \emptyset$, a configuration is a tuple $c = (\langle p, w\rangle, \Delta)$, since there is no changing rule, so there is only one possible phase. In this case, we can also write $c = \langle p, w\rangle$. Let $\mathcal{C}$ be the set of configurations of a SM-PDS. A SM-PDS defines a transition relation $\Rightarrow_{\mathcal{P}}$ between configurations as follows: Let $c = (\langle p, w\rangle, \theta)$ be a configuration, and let r be a rule in θ, then:

1) if $r \in \Delta_c$ is of the form $r = p \xhookrightarrow{(r_1,r_2)} p'$, such that $r_1 \in \theta$, then $(\langle p, w\rangle, \theta) \Rightarrow_{\mathcal{P}} (\langle p', w\rangle, \theta')$, where $\theta' = (\theta \setminus \{r_1\}) \cup \{r_2\}$. In other words, the transition rule r updates the current set of transition rules θ by removing r_1 from it and adding r_2 to it.
2) if $r \in \Delta$ is of the form $r = \langle p, \gamma\rangle \hookrightarrow \langle p', w'\rangle \in \Delta$, then $(\langle p, \gamma w\rangle, \theta) \Rightarrow_{\mathcal{P}} (\langle p', w'w\rangle, \theta)$. In other words, the transition rule r moves the control point from p to p', pops γ from the stack and pushes w' onto the stack. This transition keeps the current set of transition rules θ unchanged.

Let $\Rightarrow^*_{\mathcal{P}}$ be the transitive, reflexive closure of $\Rightarrow_{\mathcal{P}}$ and $\Rightarrow^+_{\mathcal{P}}$ be its transitive closure. An execution (a run) of $\mathcal{P}$ is a sequence of configurations $\pi = c_0c_1...$ s.t. $c_i \Rightarrow_{\mathcal{P}} c_{i+1}$ for every $i \geq 0$. Given a configuration c, the set of immediate predecessors (resp. successors) of c is $pre_{\mathcal{P}}(c) = \{c' \in \mathcal{C} : c' \Rightarrow_{\mathcal{P}} c\}$ (resp. $post_{\mathcal{P}}(c) = \{c' \in \mathcal{C} : c \Rightarrow_{\mathcal{P}} c'\}$). These notations can be generalized straightforwardly to sets of configurations. Let $pre^*_{\mathcal{P}}$ (resp. $post^*_{\mathcal{P}}$) denote the reflexive-transitive closure of $pre_{\mathcal{P}}$ (resp. $post_{\mathcal{P}}$). We remove the subscript $\mathcal{P}$ when it is clear from the context.

We suppose w.l.o.g. that rules in Δ are of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ such that $|w| \leq 2$, and that the self-modifying rules $r = p \xhookrightarrow{(r_1,r_2)} p'$ in Δ_c are such that $r \neq r_1$. Note that this is not a restriction, since for a given SM-PDS, one can compute an equivalent SM-PDS that satisfies these conditions [1] .

B. SM-PDS vs. PDS

Let $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$ be a SM-PDS. It was shown in [1] that:

1) $\mathcal{P}$ can be described by an equivalent pushdown system (PDS). Indeed, since the number of phases is finite, we can encode phases in the control point of the PDS. However, this translation is not efficient since the number of control points of the equivalent PDS is $|P| \cdot 2^{\mathcal{O}(|\Delta|+|\Delta_c|)}$.
2) $\mathcal{P}$ can also be described by an equivalent Symbolic pushdown system [24], where each SM-PDS rule is represented by a *single, symbolic* transition, where the different values of the phases are encoded in a symbolic way using relations between phases. This translation is not efficient neither since the size of the relations used in the symbolic transitions is $2^{\mathcal{O}(|\Delta|+|\Delta_c|)}$.

C. From Self-modifying Code to SM-PDS

It is shown in [1] how to describe a self-modifying binary code using a SM-PDS. The basic idea is that the control locations of the SM-PDS store the control points of the binary program and the stack mimics the program's stack. Our translation relies on the disassembler Jakstab [25] to disassemble binary code, construct the control flow graph (CFG), determine indirect jumps, compute the possible values of used variables, registers and the memory locations at each control point of program. After getting the control flow graph whose edges are equipped with disassembled instructions, we translate the CFG into a SM-PDS as described in [1]. The non self-modifying instructions of the program define the rules Δ of the SM-PDS (which are standard PDS rules), and can be obtained following the translation of [8] that models non self-modifying instructions of the program by a PDS. Self-modifying instructions are represented using self-modifying transitions Δ_c of the SM-PDS. For more details, we refer the reader to [1].

III. LTL Model-Checking of SM-PDSs

A. The linear-time temporal logic LTL

Let At be a finite set of atomic propositions. LTL formulas are defined as follows (where $A \in At$):

$$\varphi := A \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2$$

Formulae are interpreted on infinite words over 2^{At}. Let $\omega = \omega^0\omega^1...$ be an infinite word over 2^{At}. We write ω_i for the suffix of ω starting at ω^i. We denote $\omega \models \varphi$ to express that ω satisfies a formula φ:

$\omega \models A \iff A \in \omega^0$
$\omega \models \neg\varphi \iff \omega \not\models \varphi$
$\omega \models \varphi_1 \vee \varphi_2 \iff \omega \models \varphi_1$ or $\omega \models \varphi_2$
$\omega \models X\varphi \iff \omega_1 \models \varphi$
$\omega \models \varphi_1 U \varphi_2 \iff \exists i \geq 0, \omega_i \models \varphi_2$ and $\forall 0 \leq j < i, \omega_j \models \varphi_1$

The temporal operators G (globally) and F (eventually) are defined as follows: $F\varphi = (A \vee \neg A)U\varphi$ and $G\varphi = \neg F \neg \varphi$. Let $W(\varphi)$ be the set of infinite words that satisfy an LTL formula φ. It is well known that $W(\varphi)$ can be accepted by Büchi automata:

Definition 2. *A Büchi automaton $\mathcal{B}$ is a quintuple $(Q, \Gamma, \eta, q_0, F)$ where Q is a finite set of states, Γ is a finite input alphabet, $\eta \subseteq (Q \times \Gamma \times Q)$ is a set of transitions, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of accepting states. A run of $\mathcal{B}$ on a word $\gamma_0\gamma_1... \in \Gamma^\omega$ is a sequence of states $q_0q_1q_2...$ s.t. $\forall i \geq 0, (q_i, \gamma_i, q_{i+1}) \in \eta$. An infinite word ω is accepted by $\mathcal{B}$ if $\mathcal{B}$ has a run on ω that starts at q_0 and visits accepting states from F infinitely often.*

Theorem. *[26] Given an LTL formula φ, one can effectively construct a Büchi automaton $\mathcal{B}_\varphi$ which accepts $W(\varphi)$.*

B. Self Modifying Büchi Pushdown Systems

Definition 3. *A Self Modifying Büchi Pushdown System (SM-BPDS) is a tuple $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$ where $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$ is a SM-PDS, $G \subseteq P$ is a set of accepting control locations. A run π of the SM-BPDS $\mathcal{BP}$ is a run (execution) of the SM-PDS $\mathcal{P}$. π is accepting iff it infinitely often visits configurations having control locations in G.*

Let $\Rightarrow_{\mathcal{BP}}$ be the relation $\Rightarrow_{\mathcal{P}}$ ($\mathcal{P}$ is the SM-PDS underlying the SM-BPDS). Let c and c' be two configurations of the SM-BPDS $\mathcal{BP}$. The relation $\Rightarrow^r_{\mathcal{BP}}$ is defined as follows: $c \Rightarrow^r_{\mathcal{BP}} c'$ iff there exists a configuration $(\langle g, u\rangle, \theta)$, $g \in G$ s.t. $c \Rightarrow^*_{\mathcal{P}} (\langle g, u\rangle, \theta) \Rightarrow^+_{\mathcal{P}} c'$. We remove the subscript $\mathcal{BP}$ when it is clear from the context.

A head of SM-BPDS is a tuple $(\langle p, \gamma\rangle, \theta)$ where $p \in P$, $\gamma \in \Gamma$ and $\theta \subseteq \Delta \cup \Delta_c$. A head $((p, \gamma), \theta)$ is repeating if there exists $v \in \Gamma^*$ such that $(\langle p, \gamma\rangle, \theta) \Rightarrow^r_{\mathcal{BP}} (\langle p, \gamma v\rangle, \theta)$. The set of repeating heads of SM-BPDS is called $Rep_{\mathcal{BP}}$.

C. From LTL Model-Checking of SM-PDSs to the emptiness problem of SM-BPDSs

Let $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$ be a self modifying pushdown system. Let At be a set of atomic propositions. Let $\nu : P \to 2^{At}$ be a labelling function. Let $\pi = (\langle p_0, w_0\rangle, \theta_0)(\langle p_1, w_1\rangle, \theta_1)...$ be an execution of the SM-PDS $\mathcal{P}$. Let φ be an LTL formula over the set of atomic propositions At. We say that

$$\pi \models_\nu \varphi \text{ iff } \nu(p_0)\nu(p_1)\cdots \models \varphi$$

Let $(\langle p, w\rangle, \theta)$ be a configuration of $\mathcal{P}$. We say that $(\langle p, w\rangle, \theta) \models_\nu \varphi$ iff $\mathcal{P}$ has a path π starting at $(\langle p, w\rangle, \theta)$ such that $\pi \models_\nu \varphi$.

Our goal in this paper is to perform LTL model-checking for self-modifying pushdown systems. Since SM-PDSs can be translated to standard (symbolic) pushdown systems, one way to solve this LTL model-checking problem is to compute the (symbolic) pushdown system that is equivalent to the SM-PDS (see section II-B), and then apply the standard LTL model-checking algorithms on standard PDSs [24]. However, this approach is not efficient (as will be witnessed later in the experiments). Thus, we need a *direct* approach that performs LTL model-checking on the SM-PDS, without translating it to an equivalent PDS. Let then $\mathcal{B}_\varphi = (Q, 2^{At}, \eta, q_0, F)$ be a Büchi automaton that accepts $W(\varphi)$. We compute the SM-BPDS $\mathcal{BP}_\varphi = (P \times Q, \Gamma, \Delta', \Delta'_c, G)$ by performing a kind of product between the SM-PDS $\mathcal{P}$ and the Büchi automaton $\mathcal{B}_\varphi$ as follows:

1) if $r = \langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta$ and $(q, \nu(p), q') \in \eta$, then $\langle (p, q), \gamma\rangle \hookrightarrow \langle (p', q'), w\rangle \in \Delta'$. Let $prod(r)$ be the set of rules of Δ' obtained from the rule r, i.e., rules of Δ' of the form $\langle (p, q), \gamma\rangle \hookrightarrow \langle (p', q'), w\rangle$.
2) if a rule $r = p \xhookrightarrow{(r_1, r_2)} p' \in \Delta_c$ and $(q, \nu(p), q') \in \eta$, then $\{(p, q) \xhookrightarrow{(r'_1, r'_2)} (p', q'), r'_1 \in prod(r_1), r'_2 \in prod(r_2)\} \subseteq \Delta'_c$. Let $prod(r)$ be the set of rules of Δ' obtained from the rule r, i.e., rules of Δ'_c of the form $(p, q) \xhookrightarrow{(r'_1, r'_2)} (p', q')$, where $r'_1 \in prod(r_1)$, and $r'_2 \in prod(r_2)$.
3) $G = P \times F$.

We can show that:

Theorem 1. *Let $(\langle p, w\rangle, \theta)$ be a configuration of the SM-PDS $\mathcal{P}$. $(\langle p, w\rangle, \theta) \models_\nu \varphi$ iff $\mathcal{BP}_\varphi$ has an accepting run from $(\langle (p, q_0), w\rangle, prod(\theta))$ where $prod(\theta)$ is the set of rules of $\Delta \cup \Delta_c$ obtained from the rules of θ as described above.*

Thus, LTL model-checking for SM-PDSs can be reduced to checking whether an SM-BPDS has an accepting run. The rest of the paper is devoted to this problem.

IV. The Emptiness Problem of SM-BPDSs

From now on, we fix an SM-BPDS $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$. We can show that $\mathcal{BP}$ has an accepting run starting from a configuration c if and only if from c, it can reach a configuration with a repeating head:

Proposition 1. *An SM-BPDS $\mathcal{BP}$ has an accepting run starting from a configuration c if and only if there exists a repeating head $((p, \gamma), \theta)$ such that $c \Rightarrow^*_{\mathcal{BP}} (\langle p, \gamma w\rangle, \theta)$ for some $w \in \Gamma^*$.*

Thus, since there exists an efficient algorithm to compute the pre^* of SM-PDSs [1], the emptiness problem of an SM-BPDS can be reduced to computing its repeating heads.

A. The Head Reachability Graph $\mathcal{G}$

Our goal is to compute the set of repeating heads $Rep_{\mathcal{BP}}$, i.e., the set of heads $(\langle p, \gamma\rangle, \theta)$ such that there exists $v \in \Gamma^*$, $(\langle p, \gamma\rangle, \theta) \Rightarrow^r (\langle p, \gamma v\rangle, \theta)$. I.e., $(\langle p, \gamma\rangle, \theta) \Rightarrow^* (\langle p, \gamma v\rangle, \theta)$ s.t. this path goes through an accepting location in G. To this aim, we will compute a finite graph $\mathcal{G}$ whose nodes are the heads of $\mathcal{BP}$ of the form $((p, \gamma), \theta)$, where $p \in P$, $\gamma \in \Gamma$ and $\theta \subseteq \Delta \cup \Delta_c$; and whose edges encode the reachability relation betweeen these heads. More precisely, given two heads $((p, \gamma), \theta)$ and $((p', \gamma'), \theta')$, $((p, \gamma), \theta) \xrightarrow{b} ((p', \gamma'), \theta')$ is an edge of the graph $\mathcal{G}$ means that the configuration $(\langle p, \gamma\rangle, \theta)$ can reach a configuration having $(\langle p', \gamma'\rangle, \theta')$ as head, i.e., it means that there exists $v \in \Gamma^*$ s.t. $(\langle p, \gamma\rangle, \theta) \Rightarrow^* (\langle p', \gamma' v\rangle, \theta')$. Moreover, we need to keep the information whether this path visits an accepting location in G or not. This information is recorded in the label of the edge b: $b = 1$ means that the path visits an accepting location in G, i.e. that $(\langle p, \gamma\rangle, \theta) \Rightarrow^r (\langle p', \gamma' v\rangle, \theta')$. Otherwise, $b = 0$. Therefore, if the graph $\mathcal{G}$ contains a loop from a head $((p, \gamma), \theta)$ to itself such that this loop goes through an edge labelled by 1, then $((p, \gamma), \theta)$ is a repeating head. Thus, computing $Rep_{\mathcal{BP}}$ can be reduced to computing the graph $\mathcal{G}$ and finding 1-labelled loops in this graph.

More precisely, we define the head reachability graph $\mathcal{G}$ as follows:

Definition 4. *The head reachability graph $\mathcal{G}$ is a tuple $(P \times \Gamma \times 2^{\Delta \cup \Delta_c}, \{0, 1\}, \delta)$ such that $((p, \gamma), \theta) \xrightarrow{b} ((p', \gamma'), \theta')$ is an edge of δ iff:*

1) *there exists a transition* $r_c : p \xhookrightarrow{(r_1,r_2)} p' \in \theta \cap \Delta_c$, $\gamma = \gamma'$, $\theta' = \theta \setminus \{r_1\} \cup \{r_2\}$, *and* $b = 1$ *iff* $p \in G$;
2) *there exists a transition* $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \theta \cap \Delta, \theta = \theta'$ *and* $b = 1$ *iff* $p \in G$;
3) *there exists a transition* $\langle p, \gamma\rangle \hookrightarrow \langle p'', \gamma_1\gamma'\rangle \in \theta \cap \Delta$, *for* $\gamma_1 \in \Gamma$, $p'' \in P$, *s.t.* $(\langle p'', \gamma_1\rangle, \theta) \Rightarrow^*_{\mathcal{BP}} (\langle p', \epsilon\rangle, \theta')$, *and* $b = 1$ *iff* $p \in G$ *or* $(\langle p'', \gamma_1\rangle, \theta) \Rightarrow^r_{\mathcal{BP}} (\langle p', \epsilon\rangle, \theta')$

We can show that:

Theorem 2. *Let* $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$ *be a self-modifying Büchi pushdown system, and let* $\mathcal{G}$ *be its corresponding head reachability graph. A head* $((p, \gamma), \theta)$ *of* $\mathcal{BP}$ *is repeating iff* $\mathcal{G}$ *has a loop on the node* $((p, \gamma), \theta)$ *that goes through a 1-labeled edge.*

B. Labelled configurations and labelled $\mathcal{BP}$-automata

To compute $\mathcal{G}$, we need to be able to compute predecessors of configurations of the form $(\langle p', \epsilon\rangle, \theta')$, and to determine whether these predecessors were backward-reachable using some control points in G (item 3 in Definition 4). To solve this question, we will label configurations $(\langle p'', w\rangle, \theta)$ s.t. $(\langle p'', w\rangle, \theta) \Rightarrow^* (\langle p', \epsilon\rangle, \theta')$ by 1 if this path went through an accepting location in G, i.e., if $(\langle p'', w\rangle, \theta) \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$, and by 0 if not. To this aim, we define a labelled configuration as a tuple $[(\langle p, w\rangle, \theta), b]$, s.t. $(\langle p, w\rangle, \theta)$ is a configuration and $b \in \{0, 1\}$.

Multi-automata were introduced in [2], [3] to finitely represent regular infinite sets of configurations of a PDS. Since a labelled configuration $c = [(\langle p, w\rangle, \theta), b]$ of a SM-PDS involves a PDS configuration $\langle p, w\rangle$, together with the current set of transition rules (phase) θ, and a boolean b, in order to take into account the phases θ, and these new $0/1$-labels in configurations, we extend multi-automata to labelled $\mathcal{BP}$-automata as follows:

Definition 5. *Let* $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, G)$ *be a SM-BPDS. A labelled* $\mathcal{BP}$*-automaton is a tuple* $\mathcal{A} = (Q, \Gamma, T, I, F)$ *where* Γ *is the automaton alphabet,* Q *is a finite set of states,* $I \subseteq P \times 2^{\Delta \cup \Delta_c} \subseteq Q$ *is the set of initial states,* $T \subset Q \times \big((\Gamma \cup \{\epsilon\}) \times \{0, 1\}\big) \times Q$ *is the set of transitions,* $F \subseteq Q$ *is the set of final states.*

If $\big(q, [\gamma, b], q'\big) \in T$, we write $q \xrightarrow{[\gamma,b]}_T q'$. We extend this notation in the obvious way to sequences of symbols: (1) $\forall q \in Q, q \xrightarrow{[\epsilon,0]}_T q$, and (2) $\forall q, q' \in Q, \forall b \in \{0, 1\}, \forall w \in \Gamma^*$ for $w = \gamma_0 ... \gamma_{n+1}, q \xrightarrow{[w,b]}_T q'$ iff $\exists q_0, ..., q_n \in Q, b_0, ..., b_{n+1} \in \{0, 1\}, b = b_0 \vee b_1 \vee ... \vee b_{n+1}$ and $q \xrightarrow{[\gamma_0,b_0]}_T q_0 \xrightarrow{[\gamma_1,b_1]}_T q_1 \cdots q_n \xrightarrow{[\gamma_{n+1},b_{n+1}]}_T q'$. If $q \xrightarrow{[w,b]}_T q'$ holds, we say that $q \xrightarrow{[w,b]}_T q'$ and $q \xrightarrow{[\gamma_0,b_0]}_T q_0 \xrightarrow{[\gamma_1,b_1]}_T q_1 \cdots q_n \xrightarrow{[\gamma_{n+1},b_{n+1}]}_T q'$ is a path of $\mathcal{A}$.

A labelled configuration $[(\langle p, w\rangle, \theta), b]$ is accepted by the automaton $\mathcal{A}$ iff there exists a path $(p, \theta) \xrightarrow{[\gamma_0,b_0]}_T q_1 \xrightarrow{[\gamma_1,b_1]}_T q_2 \cdots q_n \xrightarrow{[\gamma_n,b_n]}_T q_{n+1}$ in $\mathcal{A}$ such that $w = \gamma_0\gamma_1 \cdots \gamma_n$, $b = b_0 \vee b_1 \vee ... \vee b_n$, $(p, \theta) \in I$, and $q_{n+1} \in F$. Let $L(\mathcal{A})$ be the set of labelled configurations accepted by $\mathcal{A}$.

C. Computing $pre^*\big((\langle p', \epsilon\rangle, \theta')\big)$

Given a configuration of the form $(\langle p', \epsilon\rangle, \theta')$, our goal is to compute a labelled $\mathcal{BP}$-automaton $\mathcal{A}_{pre^*}\big((\langle p', \epsilon\rangle, \theta')\big)$ that accepts labelled configurations of the form $[c, b]$ where c is a configuration and $b \in \{0, 1\}$ such that $c \Rightarrow^* (\langle p', \epsilon\rangle, \theta')$ (i.e., $c \in pre^*\big((\langle p', \epsilon\rangle, \theta')\big)$) and $b = 1$ iff this path went through final control points, i.e., $c \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$. Otherwise, $b = 0$. Let $p \in P$, we define $B(p) = 1$ if $p \in G$ and $B(p) = 0$ otherwise. $\mathcal{A}_{pre^*}\big((\langle p', \epsilon\rangle, \theta')\big) = (Q, \Gamma, T, I, F)$ is computed as follows: Initially, $Q = I = F = \{(p', \theta')\}$ and $T = \emptyset$. We add to T transitions as follows:

- α_1: If $r = \langle p, \gamma\rangle \hookrightarrow \langle p_1, w\rangle \in \Delta$. If there exists in T a path $(p_1, \theta) \xrightarrow{[w,b]}_T q$ (in case $|w| = 0$, we have $w = \epsilon$) with $r \in \theta$. Then, add (p, θ) to I, and $\big((p, \theta), [\gamma, B(p) \vee b], q\big)$ to T.
- α_2: if $r = p \xhookrightarrow{(r_1,r_2)} p_1 \in \Delta_c$ and there exists in T a transition $(p_1, \theta) \xrightarrow{[\gamma,b]}_T q$ with $r \in \theta$, where $\gamma \in \Gamma$. Then add (p, θ') to I, and $\big((p, \theta'), [\gamma, B(p) \vee b], q\big)$ to T, for θ' such that $\theta = (\theta' \setminus \{r_1\}) \cup \{r_2\}$.

The procedure above terminates since there is a finite number of states and phases. Note that by construction, $F = \{(p', \theta')\}$, and, since initially $Q = \{(p', \theta')\}$, states of $\mathcal{A}_{pre^*}\big((\langle p', \epsilon\rangle, \theta')\big)$ are all of the form (p, θ) for $p \in P$ and $\theta \subseteq \Delta \cup \Delta_c$.

Let us explain the intuition behind rule (α_1). Let $r = \langle p, \gamma\rangle \hookrightarrow \langle p_1, w\rangle \in \Delta$. Let $c = (\langle p_1, ww'\rangle, \theta)$ and $c' = (\langle p, \gamma w'\rangle, \theta)$. Then, if $c \Rightarrow^* (\langle p', \epsilon\rangle, \theta')$, then necessarily, $c' \Rightarrow^* (\langle p', \epsilon\rangle, \theta')$. Moreover, $c' \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$ iff either $c \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$ or $p \in G$ (i.e. $B(p) = 1$). Thus, we would like that if the automaton $\mathcal{A}_{pre^*}\big((\langle p', \epsilon\rangle, \theta')\big)$ accepts the labelled configuration $[c, b]$ (where $b = 1$ means $c \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$), then it should also accept the labelled configuration $[c', b \vee B(p)]$ ($b \vee B(p) = 1$ means $c' \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$). Thus, if the automaton $\mathcal{A}_{pre^*}\big((\langle p', \epsilon\rangle, \theta')\big)$ contains a path of the form $\pi = (p_1, \theta) \xrightarrow{[w,b_1]}_T q \xrightarrow{[w',b_2]}_T q_f$ where $q_f \in F$ that accepts the labelled configuration $[c, b]$, then the automaton should also accept the labelled configuration $[c', b \vee B(p)]$. This configuration is accepted by the run $(p, \theta) \xrightarrow{[\gamma,B(p)\vee b_1]}_T q \xrightarrow{[w',b_2]}_T q_f$ added by rule (α_1).

Rule (α_2) deals with modifying rules: Let $r = p \xhookrightarrow{(r_1,r_2)} p_1 \in \Delta_c$. Let $c = (\langle p_1, \gamma w'\rangle, \theta)$ and $c' = (\langle p, \gamma w'\rangle, \theta'')$ s.t. $\theta = \theta'' \setminus \{r_1\} \cup \{r_2\}$. Then, if $c \Rightarrow^* (\langle p', \epsilon\rangle, \theta')$, then necessarily, $c' \Rightarrow^* (\langle p', \epsilon\rangle, \theta')$. Moreover, $c' \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$ iff either $c \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$ or $p \in G$ (i.e. $B(p) = 1$). Thus, we need to impose that if the automaton $\mathcal{A}_{pre^*}\big((\langle p', \epsilon\rangle, \theta')\big)$ contains a path of the form $(p_1, \theta) \xrightarrow{[\gamma,b_1]}_T q \xrightarrow{[w',b_2]}_T q_f$ (where $q_f \in F$) that accepts the labelled configuration $[c, b], b = b_1 \vee b_2$ ($b = 1$ means $c \Rightarrow^r (\langle p', \epsilon\rangle, \theta')$), then necessarily, the automaton $\mathcal{A}_{pre^*}\big((\langle p', \epsilon\rangle, \theta')\big)$ should also accept the labelled configuration $[c', b \vee B(p)]$. This configuration is accepted by the run $(p, \theta'') \xrightarrow{[\gamma,B(p)\vee b_1]}_T q \xrightarrow{[w',b_2]}_T q_f$ added by rule (α_2).

We can show that:

Lemma 1. *Let* $p, p'' \in P$ *and* $\theta, \theta'' \subseteq \Delta \cup \Delta_c$. *Let* $w \in \Gamma^*$ *and* $b \in \{0,1\}$. *If a path* $(p,\theta) \xrightarrow{[w,b]}_T (p'',\theta'')$ *is in* $\mathcal{A}_{pre^*}\big((\langle p',\epsilon\rangle,\theta')\big)$, *then* $(\langle p,w\rangle,\theta) \Rightarrow^* (\langle p'',\epsilon\rangle,\theta'')$. *Moreover, if* $b=1$, *then* $(\langle p,w\rangle,\theta) \Rightarrow^r (\langle p'',\epsilon\rangle,\theta'')$.

Lemma 2. *If there is a labelled configuration* $[(\langle p,w\rangle,\theta),b]$ *such that* $(\langle p,w\rangle,\theta) \Rightarrow^* (\langle p',\epsilon\rangle,\theta')$, *then there is a path* $(p,\theta) \xrightarrow{[w,b]}_T (p',\theta')$ *in* $\mathcal{A}_{pre^*}\big((\langle p',\epsilon\rangle,\theta')\big)$. *Moreover, if* $(\langle p,w\rangle,\theta) \Rightarrow^r (\langle p',\epsilon\rangle,\theta')$, *then* $b=1$.

From these two lemmas, we get:

Theorem 3. *Let* $[c,b]$ *be a labelled configuration. Then* $[c,b]$ *is in* $L(\mathcal{A}_{pre^*}\big((\langle p',\epsilon\rangle,\theta')\big)$ *iff* $c \in pre^*\big((\langle p',\epsilon\rangle,\theta')\big)$. *Moreover,* $c \Rightarrow^r (\langle p',\epsilon\rangle,\theta')$ *iff* $b=1$.

D. Computing the Head Reachability Graph $\mathcal{G}$

Based on the definition of the Head Reachability Graph $\mathcal{G}$, and on Theorem 3, we can compute $\mathcal{G}$ as follows. Initially, $\mathcal{G}$ has no edges.

- α'_1: if $r_c : p \xhookrightarrow{(r_1,r_2)} p' \in \Delta_c$, then for every phase θ such that $r_c \in \theta$ and every $\gamma \in \Gamma$, we add the edge $((p,\gamma),\theta) \xrightarrow{B(p)} ((p',\gamma),\theta_0)$ to the graph $\mathcal{G}$, where $\theta_0 = \theta \setminus \{r_1\} \cup \{r_2\}$.
- α'_2: if $r : \langle p,\gamma\rangle \hookrightarrow \langle p_0,\gamma_0\rangle \in \Delta$, then for every phase θ such that $r \in \theta$, we add the edge $((p,\gamma),\theta) \xrightarrow{B(p)} ((p_0,\gamma_0),\theta)$ to the graph $\mathcal{G}$.
- α'_3: if $r : \langle p,\gamma\rangle \hookrightarrow \langle p_0,\gamma_0\gamma'\rangle \in \Delta$, then for every phase θ such that $r \in \theta$, we add to the graph $\mathcal{G}$ the edge $((p,\gamma),\theta) \xrightarrow{B(p)} ((p_0,\gamma_0),\theta)$. Moreover, for every control point $p' \in P$ and phase θ' such that $\mathcal{A}_{pre^*}\big((\langle p',\epsilon\rangle,\theta')\big)$ contains a transition of the form $t = (p_0,\theta) \xrightarrow{[\gamma_0,b]}_T (p',\theta')$, we add to the graph $\mathcal{G}$ the edge $((p,\gamma),\theta) \xrightarrow{b \vee B(p)} ((p',\gamma'),\theta')$.

Items α'_1 and α'_2 are obvious. They respectively correspond to item 1 and item 2 of Definition 4 (since $B(p)=1$ iff $p \in G$). Item α'_3 is based on Lemma 1 and on item 3 of Definition 4. Indeed, it follows from Lemma 1 that $\mathcal{A}_{pre^*}\big((\langle p',\epsilon\rangle,\theta')\big)$ contains a transition of the form $(p_0,\theta) \xrightarrow{[\gamma_0,b]}_T (p',\theta')$ implies that $(\langle p_0,\gamma_0\rangle,\theta) \Rightarrow^* (\langle p',\epsilon\rangle,\theta')$, and if $b=1$, then $(\langle p_0,\gamma_0\rangle,\theta) \Rightarrow^r (\langle p',\epsilon\rangle,\theta')$. Thus, in this case, the edge $((p,\gamma),\theta) \xrightarrow{b \vee B(p)} ((p',\gamma'),\theta')$ is added to $\mathcal{G}$ (item 3 of Definition 4) since $\langle p,\gamma\rangle \hookrightarrow \langle p_0,\gamma_0\gamma'\rangle \in \Delta$.

Remark 1. *As described above, computing the graph* $\mathcal{G}$ *is based on determining whether* $(\langle p_0,\gamma_0\rangle,\theta) \Rightarrow^r (\langle p',\epsilon\rangle,\theta')$, *for* $p', p_0 \in P$, $\gamma_0 \in \Gamma$, *and* θ, θ' *phases. In this section, we showed how to answer this question using a backward approach based on computing the* labelled pre^* *of configurations of the form* $(\langle p',\epsilon\rangle,\theta')$. *We can also apply a forward approach based on computing the* labelled $post^*$ *of configurations of the form* $(\langle p_0,\gamma_0\rangle,\theta)$. *Computing the* labelled $post^*$ *can be done using labelled* $\mathcal{BP}$*-automata in the same spirit as above by extending the "standard"* $post^*$ *saturation computation of SM-PDSs presented in [1] in order to take into account the* $0/1$*-labels as done above for the* labelled pre^*.

V. Experiments

A. Our approach vs. standard LTL for PDSs

We implemented our approach in a tool and we compared its performance against the approaches that consist in translating the SM-PDS to an equivalent standard (or symbolic) PDS, and then applying the standard LTL model checking algorithms implemented in the PDS model-checker tool Moped [24]. All our experiments were run on Ubuntu 16.04 with a 2.7 GHz CPU, 2GB of memory.

$\lvert\Delta\rvert : \lvert\Delta_c\rvert$	$\lvert\delta\rvert$	SM-PDS	PDS	Result	Total	Symbolic PDS	$Result_1$	$Total_1$
5 : 2	15	**0.07s**	0.09s	0.01s	0 .10s	0.08s	0.00s	0.08s
5 : 3	8	**0.06s**	0.08s	0.01s	0.09s	0.09s	0.00s	0.09s
11 : 4	8	**0.16s**	0.13s	0.05s	0.18s	0.10s	0.00s	0.10s
5 : 3	10	**0.06s**	0.15s	0.01s	0.16s	0.09s	0.00s	0.09s
110 : 4	8	**0.34s**	186.10s	0.79s	186.99s	0.35s	0.00s	0.35s
255 : 8	8	**0.39s**	281.02s	0.94s	281.96s	4.82s	0.05s	4.87s
255 : 8	10	**0.42s**	281.02s	0.97s	281.99s	4.82s	0.06s	4.88s
110 : 4	15	**0.28s**	186.10s	1.05s	187.15s	0.35s	0.06s	0.41s
255 : 8	15	**0.46s**	281.02s	1.92s	282.94s	4.82s	0.08s	4.90s
110 : 4	20	**0.37s**	186.10s	1.05s	187.15s	0.35s	0.06s	0.41s
255 : 8	20	**0.55s**	281.02s	1.97s	282.99s	4.82s	0.17s	4.99s
255 : 8	25	**0.59s**	281.02s	1.23s	282.99s	4.82s	0.24s	5.36s
2059 : 7	8	**0.86s**	19525.01s	20.71s	19545.72s	20.70s	error	-
2059 : 9	8	**1.49s**	19784.7s	79.12s	19863.32	128.12s	error	-
2059 : 11	8	**3.73s**	30011.67s	168.15s	30179.82s	261.07s	error	-
2059 : 11	28	**6.88s**	30011.67s	169.55s	30180.22s	261.07s	error	-
3050 : 10	8	**5.21s**	39101.57s	killed	-	438.27s	error	-
3090 : 10	8	**5.86s**	40083.07s	killed	-	438.69s	error	-
3050 : 10	20	**7.24s**	39101.57s	killed	-	438.27s	error	-
3090 : 10	30	**8.38s**	40083.07s	killed	-	438.69s	error	-
3090 : 10	25	**8.89s**	40083.07s	killed	-	438.69s	error	-
4050 : 10	8	**9.21s**	81408.91s	killed	-	699.19s	error	-
4050 : 10	28	**11.64s**	81408.91s	killed	-	699.19s	error	-
4058 : 11	8	**9.83s**	93843.37s	killed	-	802.07s	error	-
4058 : 11	25	**13.59s**	93843.37s	killed	-	802.07s	error	-
5050 : 11	8	**10.34s**	173943.37s	killed	-	921.16s	error	-
5090 : 11	8	**10.52s**	179993.54s	killed	-	929.32s	error	-
5090 : 11	10	**12.89s**	179993.54s	killed	-	929.32s	error	-
6090 : 11	8	**13.49s**	190293.64s	killed	-	1002.73s	error	-
6090 : 11	10	**15.81s**	190293.64s	killed	-	1002.73s	error	-
6090 : 11	40	**32.39s**	190293.64s	killed	-	1002.73s	error	-
7090 : 11	25	**39.86s**	198932.32s	killed	-	1092.28s	error	-
7090 : 11	30	**43.24s**	198932.32s	killed	-	1092.28s	error	-
9090 : 11	8	**29.98s**	199987.98s	killed	-	1128.19s	error	-
9090 : 11	20	**45.29s**	199987.98s	killed	-	1128.19s	error	-
10050 : 12	8	**48.53s**	2134587.14s	killed	-	1469.28s	error	-
10050 : 12	25	**59.69s**	2134587.14s	killed	-	1469.28s	error	-
10050 : 12	30	**61.42s**	2134587.14s	kille d	-	1469.28s	error	-
10150 : 12	35	**64.17s**	2134633.28s	killed	-	1469.28s	error	-
10150 : 14	8	**58.34s**	2181975.64s	killed	-	2849.96s	error	-
10150 : 14	40	**82.72s**	2181975.64s	killed	-	2849.96s	error	-
10150 : 12	40	**76.61s**	2134633.28s	killed	-	1469.28s	error	-
10150 : 16	45	**89.83s**	2211008.82s	killed	-	3665.59s	error	-
10150 : 12	60	**97.56s**	2134633.28s	killed	-	1469.28s	error	-
10150 : 12	65	**105.89s**	2134633.28s	killed	-	1469.28s	error	-
10150 : 16	65	**134.45s**	2211008.82s	killed	-	3665.59s	error	-
10180 : 16	65	**175.29s**	2134643.52s	killed	-	3689.83s	error	-
10180 : 16	78	**214.36s**	2134643.52s	killed	-	3689.83s	error	-

TABLE I: Our approach vs. standard LTL for PDSs

To perform the comparision, we randomly generate several SM-PDSs and LTL formulas of different sizes. The results (CPU Execution time) are shown in Table I. **Column** $|\Delta| : |\Delta_c|$ is the size of SM-PDS. **Column** $|\delta|$ gives the size of the transitions of the Büchi automaton generated from the LTL formula (using the tool LTL2BA[27]). **Column** *SM-PDS*

gives the cost of our direct algorithm presented in this paper. **Column** *PDS* shows the cost it takes to get the equivalent PDS from the SM-PDS. **Column** *Result* reports the cost it takes to run the LTL PDS model-checker Moped [24] for the PDS we got. **Column** *Total* is the total cost it takes to translate the SM-PDS into a PDS and then apply the standard LTL model checking algorithm of Moped (Total=PDS+Result). **Column** *Symbolic PDS* reports the cost it takes to get the equivalent Symbolic PDS from the SM-PDS. **Column** $Result_1$ is the cost to run the Symbolic PDS LTL model-checker Moped. **Column** $Total_1$ is the total cost it takes to translate the SM-PDS into a symbolic PDS and then apply the standard LTL model checking algorithm of Moped. You can see that our direct algorithm (**Column** *SM-PDS*) is much more efficient than translating the SM-PDS to an equivalent (symbolic) PDS, and then run the standard LTL model-checker Moped. **Translating the SM-PDS to a standard PDS may take more than 20 days, whereas our direct algorithm takes only a few seconds.** Moreover, since the obtained standard (symbolic) PDS is huge, Moped failed to handle several cases (the time limit that we set for Moped is 20 minutes), whereas our tool was able to deal with all the cases in only a few seconds.

B. Malicious Behavior Detection on Self-Modifying Code

1) Specifying Malicious Behaviors using LTL.: As described in [10], several malicious behaviors can be described by LTL formulas. We give in what follows three examples of such malicious behaviors and show how they can be described by LTL formulas:

Registry Key Injecting: In order to get started at boot time, many malwares add themselves into the registry key listing. This behavior is typically implemented by first calling the API function GetModuleFileNameA to retrieve the path of the malware's executable file. Then, the API function RegSetValueExA is called to add the file path into the registry key listing. This malicious behavior can be described in LTL as follows:

$$\phi_{rk} = \mathbf{F}\big(call\ GetModuleFileNameA \wedge \mathbf{F}(\ call\ RegSetValueExA)\big)$$

This formula expresses that if a call to the API function GetModuleFileNameA is followed by a call to the API function RegSetValueExA, then probably a malware is trying to add itself into the registry key listing.

Data-Stealing: Stealing data from the host is a popular malicious behavior that intend to steal any valuable information including passwords, software codes, bank information, etc. To do this, the malware needs to scan the disk to find the interesting file that he wants to steal. After finding the file, the malware needs to locate it. To this aim, the malware first calls the API function GetModuleHandleA to get a base address to search for a location of the file. Then the malware starts looking for the interesting file by calling the API function FindFirstFileA. Then the API functions CreateFileMappingA and MapViewOfFile are called to access the file. Finally, the specific file can be copied by calling the API function CopyFileA. Thus, this data-stealing malicious behavior can be described by the following LTL formula as follows:

$$\phi_{ds} = \mathbf{F}(call\ GetModuleHandleA\ \wedge \mathbf{F}(call\ FindFirstFileA \wedge \mathbf{F}\ (call\ CreateFileMappingA\ \wedge \mathbf{F}\ (call\ MapViewofFile \wedge \mathbf{F}\ call\ CopyFileA))))$$

Spy-Worm: A spy worm is a malware that can record data and send it using the Socket API functions. For example, Keylogger is a spy worm that can record the keyboard states by calling the API functions GetAsyKeyState and GetKeyState and send that to the specific server by calling the socket function sendto. Another spy worm can also spy on the I/O device rather than the keyboard. For this, it can use the API function GetRawInputData to obtain input from the specified device, and then send this input by calling the socket functions send or sendto. Thus, this malicious behavior can be described by the following LTL formula:

$$\phi_{sw} = \mathbf{F}\big((call\ GetAsyncKeyState \vee call\ GetRawInputData) \wedge \mathbf{F}(call\ sendto \vee call\ send)\big)$$

Appending virus: An appending virus is a virus that inserts a copy of its code at the end of the target file. To achieve this, since the real OFFSET of the virus' variables depends on the size of the infected file, the virus has to first compute its real absolute address in the memory. To perform this, the virus has to call the sequence of instructions: l_1: call f; l_2: ....; f: pop eax;. The instruction call f will push the return address l_2 onto the stack. Then, the pop instruction in f will put the value of this address into the register eax. Thus, the virus can get its real absolute address from the register eax. This malicious behavior can be described by the following LTL formula:

$$\phi_{av} = \bigvee \mathbf{F}\Big(call \wedge \mathbf{X}(\text{top-of-stack} = a) \wedge \mathbf{G}\neg\big(ret \wedge(\text{top-of-stack} = a)\big)\Big)$$

where the $\bigvee$ is taken over all possible return addresses a, and top-of-stack=a is a predicate that indicates that the top of the stack is a. The subformula $call \wedge \mathbf{X}(\text{top-of-stack} = a)$ means that there exists a procedure call having a as return address. Indeed, when a procedure call is made, the program pushes its corresponding return address a to the stack. Thus, at the next step, a will be on the top of the stack. Therefore, the formula above expresses that there exists a procedure call having a as return address, such that there is no ret instruction which will return to a.

Note that this formula uses predicates that indicate that the top of the stack is a. Our techniques work for this case as well:

Example	Size	Result	$cost$	Example	Size	Result	$cost$	Example	Size	Result	$cost$
Tanatos.b	12315	Yes	16.261s	Netsky.c	45	Yes	0.002s	Win32.Happy	23	Yes	0.042s
Netsky.a	45	Yes	0.047s	Mydoom.c	155	Yes	0.014s	MyDo om-N	16980	Yes	30.231s
Mydoom.y	26902	Yes	12.462s	Mydoom.j	22355	Yes	11.262s	klez-N	6281	Yes	3.252s
klez.c	30	Yes	0.039s	Mydoom.v	5965	Yes	3.971s	Netsky.b	45	Yes	0.057s
Repah.b	221	Yes	2.428s	Gibe.b	5358	Yes	4.229s	Magistr.b	4670	Yes	3.699s
Netsky.d	45	Yes	0.083s	Ardurk.d	1913	Yes	0.482s	klez.f	27	Yes	0.054s
Kelino.1	495	Yes	0.326s	Kipis.t	20378	Yes	25.345s	klez.d	31	Yes	0.085s
Kelino.g	470	Yes	0.672s	Plage.b	395	Yes	0.291s	Urbe.a	123	Yes	0.376s
klez.e	27	Yes	0.094s	Magistr.b	4670	Yes	3.987s	Magistr.a.poly	36989	Yes	49.863s
Mydoom.M@mm	5965	Yes	5.633s	MyDoom.54464	5935	Yes	5.939s	MyDoom.N!worm	5970	Yes	6.152s
Win32.Runouce	51678	Yes	92.692s	Win32.Chur.A	51895	Yes	98.161s	Win32.CNHacker.C	51095	Yes	94.952s
Win32.Mydoom!O	215	Yes	0.481s	Mydoom.o@MM!zip	257	Yes	0.298s	W.Mydoom.kZ2L	228	Yes	0.729s
Mydoom-EG [Trj]	230	Yes	0.242s	Email.Worm.W32!c	220	Yes	0.249s	W32.Mydoom.L	235	Yes	0.288s
Worm.Mydoom-5	228	Yes	0.307s	Mydoom.CJDZ-5239	225	Yes	0.392s	Mydoom.DN.worm	220	Yes	0.299s
Win32.Mydoom.R	230	Yes	0.322s	Win32 .Mydoom.dlnpqi	235	Yes	0.296s	Mydoom.o@MM!zip	235	Yes	0.403s
Sramota.avf	240	Yes	0.383s	BehavesLike.Mydoom	238	Yes	0.278s	Win32.Mydoom.288	248	Yes	0.410s
Mydoom.ACQ	19210	Yes	39.662s	Mydoom.ba	19423	Yes	38.269s	Mydoom.ftde	19495	Yes	39.583s
Win32.Yanz	2250	Yes	4.357s	Yanzi.QTQX-0894	2120	Yes	4.109s	Win32.Yanz.a	2410	Yes	4.465s
Win32.Skybag	4180	Yes	6.891s	Skybag.A	4310	Yes	6.205s	Netsky.ah@MM	4480	Yes	6.991s
Skybag.b	4955	Yes	6.892s	Worm.Skybag-1	4820	Yes	7.119s	Win32.Agent.R	4490	Yes	7.898s
Skybag [Wrm]	4985	Yes	7.482s	Skybag.Dvgb	4830	Yes	7.564s	Netsky.CI.worm	4550	Yes	7.180s
Agent.xpro	533	Yes	0.352s	Vilsel.lhb	15036	Yes	4.972s	Generic.2026199	433	Yes	3.489s
Vilsel.lhb	15036	Yes	26.962s	Generic.DF	5358	Yes	7.821s	LdPinch.aoq	7695	Yes	6.290s
Jorik	837	Yes	4.159s	Bugbear-B	9278	Yes	17.737s	Tanatos.O	9284	Yes	21.481s
Gen.2	1510	Yes	5.632s	Gibe.b	5358	Yes	9.615s	Generic26.AXCN	837	Yes	3.792s
Androm	95	Yes	0.028s	Ardurk.d	1913	Yes	3.679s	Generic.128 61	30183	Yes	72.264s
LdPinch.by	970	Yes	4.092s	Generic.2026199	433	Yes	2.402s	LdPinch.arr	1250	Yes	1.848s
Generic.12861	30183	Yes	88.294s	Generic.18017273	267	Yes	0.192s	LdPinch.mg	5957	Yes	9.297s
Script.489524	522	Yes	1.458s	Generic.DF	5358	Yes	8.291s	Zafi	433	Yes	1.028s
GenericKD4047614	3495	Yes	4.646s	Win32.Agent.es	3500	Yes	6.083s	W32.HfsAutoB.	3398	Yes	5.092s
Trojan.Sivis-1	5351	Yes	7.029s	Win32.Siggen.28	5440	Yes	6.998s	Trojan/Cosmu.isk.	5345	Yes	6.273s
Trojan.17482-4	381	Yes	1.495s	Delphi.Gen	375	Yes	1.948s	Trojan.b5ac.	370	Yes	2.089s
LDPinch.400	1783	Yes	4.893s	PSW.LdPinch.plt	1808	Yes	5.088s	PSW.Pinch.1	1905	Yes	5.757s
LdPinch.BX.DLL	2010	Yes	6.965s	LdPinch.fmye	1845	Yes	6.194s	LdPinch.Win32.5558	2015	Yes	6.907s
TrojanSpy.Lydra.a	3450	Yes	8.289s	Trojan.StartPage	2985	Yes	5.982s	PSWTroj.LdPinch.au	2985	Yes	6.198s
TrojanSpy.Zbot	610	Yes	1.610s	LDPinch.10639	605	Yes	1.185s	SillyProxy.AM	590	Yes	1.882s
LdPinch.mj!c	590	Yes	4.5345s	LdPinch.H.gen!Eldorado	605	Yes	3.955s	Generic!BT	615	Yes	2.085s
LdPnch-Fam	195	Yes	1.440s	Troj.LdPinch.er	205	Yes	2.529s	LdPinch.Gen.3	210	Yes	1.482s
Win32.Malware.wsc	150	Yes	2.843s	malicious.7aa9fd	185	Yes	2.189s	WS.LDPinch.400	195	Yes	1.898s
calculation.exe	9952	No	18.352s	cisvc.exe	4105	No	3.631s	simple.exe	52	No	0.001s
shutdown.exe	2529	No	0.397s	loop.exe	529	No	9.249s	cmd.exe	1324	No	13.466s
notepad.exe	10529	No	24.583s	java.exe	800	No	15.852s	java.exe	21324	No	42.373s
sort.exe	8529	No	29.789s	bibDesk.exe	32800	No	50.279s	interface.exe	1005	No	8.462s
ipv4.exe	968	No	4.186s	TextWrangler.exe	14675	No	45.221s	sogou.exe	45219	No	55.259s
game.exe	34325	No	82.424s	cycle.tex	9014	No	42.555s	calender.exe	892	No	35.039s
SdBot.zk	3430	Yes	23.242s	Virus.Gen	661	Yes	9.437s	AutoRun.PR	240	Yes	4.181s
Adon.1703	37	Yes	0.358s	Adon.1559	37	Yes	0.255s	Spam.Tedroo.AB	487	Yes	0.924s
Akez	273	Yes	0.136s	Alcaul.d	845	Yes	0.165s	Alaul.c	355	Yes	0.109s
Virus.Win32.klk	5235	Yes	15.863s	Virus.Win32.Agent	5340	Yes	15.968s	Hoax.Gen	5455	Yes	13.569s
eHeur.Virus02	420	Yes	4.985s	Akez.11255	440	Yes	3.985s	Akez.Win32.1	455	Yes	4.008s
W95/Kuang	435	Yes	2.985s	Radar01.Gen	465	Yes	4.005s	Akez.Win32.5	490	Yes	3.958s
Haharin.A	210	Yes	1.462s	fsAutoB.F026	245	Yes	1.698s	Haharin.dr	235	Yes	1.558s
Tanatos.b	12315	Yes	16.261s	Netsky.c	45	Yes	0.002s	Win32.Happy	23	Yes	0.042s
Netsky.a	45	Yes	0.047s	Mydoom.c	155	Yes	0.014s	MyDoom-N	16980	Yes	30.231s
Mydoom.y	26902	Yes	12.462s	Mydoom.j	22355	Yes	11.262s	klez-N	6281	Yes	3.252s
klez.c	30	Yes	0.039s	Mydoom.v	5965	Yes	3.971s	Netsky.b	45	Yes	0.057s
Repah.b	221	Yes	2.428s	Gibe.b	5358	Yes	4.229s	Magistr.b	4670	Yes	3.699s
Netsky.d	45	Yes	0.083s	Ardurk.d	1913	Yes	0.482s	klez.f	27	Yes	0.054s
Kelino.1	495	Yes	0.326s	Kipis.t	20378	Yes	25.345s	klez.d	31	Yes	0.085s
Kelino.g	470	Yes	0.672s	Plage.b	395	Yes	0.291s	Urbe.a	123	Yes	0.376s
klez.e	27	Yes	0.094s	Magistr.b	4670	Yes	3.987s	Magistr.a.poly	36989	Yes	49.863s
Mydoom.M@mm	5965	Yes	5.633s	MyDoom.54464	5935	Yes	5.939s	MyDoom.N!worm	5970	Yes	6.152s
Win32.Runouce	51678	Yes	92.692s	Win32.Chur.A	51895	Yes	98.161s	Win32.CNHacker.C	51095	Yes	94.952s
Win32.Mydoom!O	215	Yes	0.481s	Mydoom.o@MM!zip	257	Yes	0.298s	W.Mydoom.kZ2L	228	Yes	0.729s
Mydoom-EG [Trj]	230	Yes	0.242s	Email.Worm.W32!c	220	Yes	0.249s	W32.Mydoom.L	235	Yes	0.288s
Worm.Mydoom-5	228	Yes	0.307s	Mydoom.CJDZ-5239	225	Yes	0.392s	Mydoom.DN.worm	220	Yes	0.299s

TABLE II: Partial Experimental Results

Example	Size	LTL	Multiple pre^*	Example	Size	LTL	Multiple pre^*	Example	Size	LTL	Multiple pre^*
Tanatos.b	12315	16.261s	46.635s	Netsky.c	45	0.002s	0.092s	Win32.Happy	23	0.042s	0.075s
Netsky.a	45	0.047s	0.085s	Mydoom.c	155	0.014s	0.206s	MyDoom-N	16980	30.231s	98.418s
Mydoom.y	26902	12.462s	102.559s	Mydoom.j	22355	11.262s	111.617s	klez-N	6281	3.252s	78.419s
klez.c	30	0.039s	0.088s	Mydoom.v	5965	3.971s	83.988s	Netsky.b	45	0.057s	0.183s
Repah.b	221	2.428s	8.852s	Gibe.b	5358	4.229s	17.239s	Magistr.b	4670	3.699s	93.818s
Netsky.d	45	0.083s	0.123s	Ardurk.d	1913	0.482s	3.212s	klez.f	27	0.054s	4.518s
Kelino.l	495	0.326s	5.468s	Kipis.t	20378	23.345s	48.689s	klez.d	31	0.085s	0.291s
Kelino.g	470	0.672s	3.446s	Plage.b	395	0.291s	3.138s	Urbe.a	123	0.376s	2.981s
klez.e	27	0.094s	0.482s	Magistr.b	4670	3.987s	53.235s	Magistr.a.poly	36989	49.863s	159.195s
Adon.1703	37	0.358s	0.884s	Adon.1559	37	0.255s	4.088s	Spam.Tedroo.AB	487	0.924s	4.894s
Akez	273	0.136s	1.863s	Alcaul.d	845	0.165s	0.392s	Alaul.c	355	0.109s	5.757s
Haharin.A	210	1.462s	4.318s	fsAutoB.F026	245	1.698s	4.503s	Haharin.dr	235	1.558s	4.312s
LdPinch.BX.DLL	2010	6.965s	8.128s	LdPinch.fmye	1845	6.194s	9.232s	LdPinch.Win32.5558	2015	6.907s	8.981s
LdPinch-15	580	1.008s	3.957s	LdPinch.e	578	1.185s	3.392s	Win32/Toga!rfn	590	2.023s	3.978s
Tanatos.b	12315	16.261s	46.635s	Netsky.c	45	0.002s	0.092s	Win32.Happy	23	0.042s	0.075s
Netsky.a	45	0.047s	0.085s	Mydoom.c	155	0.014s	0.206s	MyDoom-N	16980	30.231s	98.418s
Mydoom.y	26902	12.462s	102.559s	Mydoom.j	22355	11.262s	111.617s	klez-N	6281	3.252s	78.419s
klez.c	30	0.039s	0.088s	Mydoom.v	5965	3.971s	83.988s	Netsky.b	45	0.057s	0.183s
Repah.b	221	2.428s	8.852s	Gibe.b	5358	4.229s	17.239s	Magistr.b	4670	3.699s	93.818s
Netsky.d	45	0.083s	0.123s	Ardurk.d	1913	0.482s	3.212s	klez.f	27	0.054s	4.518s
Kelino.l	495	0.326s	5.468s	Kipis.t	20378	23.345s	48.689s	klez.d	31	0.085s	0.291s
Kelino.g	470	0.672s	3.446s	Plage.b	395	0.291s	3.138s	Urbe.a	123	0.376s	2.981s
klez.e	27	0.094s	0.482s	Magistr.b	4670	3.987s	53.235s	Magistr.a.poly	36989	49.863s	159.195s
Mydoom-EG[Trj]	230	0.242s	6.172s	Email.W32!c	220	0.249s	5.946s	W32.Mydoom.L	235	0.288s	6.452s
Mydoom.5	228	0.307s	8.163s	Mydoom.cjdz5239	225	0.392s	9.968s	Mydoom.DN.worm	220	0.299s	8.928s
Mydoom.R	230	0.322s	9.086s	Win32.Mydoom	235	0.296s	7.985s	Mydoom.o@MM!zip	235	0.403s	10.323s
Mydoom.M@mm	5965	5.633s	108.129s	MyDoom.54464	5935	5.939s	94.026s	MyDoom.N	5970	6.152s	86.468s
Sramota.avf	240	0.383s	2.691s	Mydoom	238	0.278	2.749s	Win32.Mydoom.288	248	0.410s	2.983s
Win32.Runouce	51678	92.692s	248.146s	Win32.Chur.A	51895	98.161s	298.047s	Win32.CNHacker	51095	94.952s	245.452s
Win32.Skybag	4180	6.891s	13.739s	Skybag.A	4310	6.205s	15.452s	Netsky.ah@MM	4480	6.991s	16.018s
Adon.1703	37	0.358s	0.884s	Adon.1559	37	0.255s	4.088s	Spam.Tedroo.AB	487	0.924s	4.894s
Akez	273	0.136s	1.863s	Alcaul.d	845	0.165s	0.392s	Alaul.c	355	0.109s	5.757s
Haharin.A	210	1.462s	4.318s	fsAutoB.F026	245	1.698s	4.503s	Haharin.dr	235	1.558s	4.312s
LdPinch.BX.DLL	2010	6.965s	8.128s	LdPinch.fmye	1845	6.194s	9.232s	LdPinch..5558	2015	6.907s	8.981s
LdPinch-15	580	1.008s	3.957s	LdPinch.e	578	1.185s	3.392s	Win32/Toga!rfn	590	2.023s	3.978s
LdPinch.by	970	4.092s	11.327s	Generic.2026199	433	2.402s	9.614s	LdPinch.arr	1250	1.848s	9.986s
LdPnch-Fam	195	1.440s	4.097s	Troj.LdPinch.er	205	2.529s	6.154s	LdPinch.Gen.3	210	1.482s	4.973s
Androm	95	0.028s	0.192s	Ardurk.d	1913	3.679s	5.588s	Generic.12861	30183	72.264s	224.809s
Jorik	837	4.159s	11.733s	Bugbear-B	9278	17.737s	52.549s	Tanatos.O	9284	21.481s	79.773s

TABLE III: Multiple pre^* v.s. our direct LTL model-checking algorithm

it suffices to encode the top of the stack in the control points of the SM-PDS. Our implementation works for this case as well and can handle appending viruses.

2) Applying our tool for malware detection.: We applied our tool to detect several malwares. We use the unpack tool unpacker [28] to handle packers like UPX, and we use Jakstab [25] as disassembler. We consider 160 malwares from the malware library VirusShare [29], 184 malwares from the malware library MalShare [30], 288 email-worms from VX heaven [31] and 260 new malwares generated by NGVCK, one of the best malware generators. We also choose 19 benign samples from Windows XP system. We consider self-modifying versions of these programs. In these versions, the malicious behaviors are unreachable if the semantics of the self-modifying instructions are not taken into account, i.e., if the self-modifying instructions are considered as "standard" instructions that do not modify the code, then the malicious behaviors cannot be reached. To check this, we model such programs in two ways:

1) First, we take into account the self-modifying instructions and model these programs using SM-PDSs as described in Section II-C. Then, we check whether these SM-PDSs satisfy at least one of the malicious LTL formulas presented above. If yes, the program is declared as malicious, if not, it is declared as benign. Our tool was able to detect all the 892 self-modifying malwares as malicious, and to determine that benign programs are benign. We report in Table II some of the results we obtained. **Column** *Size* is the number of control locations, **Column** *Result* gives the result of our algorithm: **Yes** means malicious and **No** means benign; and **Column** *cost* gives the cost to apply our LTL model-checker to check one of the LTL properties described above.
2) Second, we abstract away the self-modifying instructions and proceed as if these instructions were not self-modifying. In this case, we translate the binary codes to standard pushdown systems as described in [8]. By using PDSs as models, none of the malwares that we consider was detected as malicious, whereas, as reported in Table II, using self-modifying PDSs as models, and applying our LTL model-checking algorithm allowed to detect all the 892 malwares that we considered.

Remark 2. *Note that checking the formulas ϕ_{rk}, ϕ_{ds}, and ϕ_{sw} could be done using multiple pre^* queries on SM-PDSs using the pre^* algorithm of [1]. However, this would be less efficient than performing our direct LTL model-checking algorithm, as shown in Table III where* ***Column*** Size *gives the number of control locations,* ***Column*** LTL *gives the time of applying our LTL model-checking algorithm; and* ***Column*** Multiple pre^* *gives the cost of applying multiple pre^* on SM-PDSs to check the properties ϕ_{rk}, ϕ_{ds}, and ϕ_{sw}. It can be seen that applying*

our tool	McAfee	Norman	BitDefender	Kinsoft	Avira	eScan	Kaspersky	Qihoo360	Baidu	Avast	Symantec
100%	24.8%	19.5%	31.2%	9.7%	34.1%	21.9%	53.1%	51.7%	1.4%	68.3%	82.4%

TABLE IV: Detection rate: Our tool vs. well known antiviruses

our direct *LTL model checking algortihm is more efficient. Furthermore, the appending virus formula ϕ_{av} cannot be solved using multiple pre^* queries. Our direct LTL model-checking algorithm is needed in this case. Note that some of the malwares we considered in our experiments are appending viruses. Thus, our algorithm and our implementation are crucial to be able to detect these malwares.*

3) Comparision with well-known antiviruses.: We compare our tool against well-known and widely used antiviruses. Since known antiviruses update their signature database as soon as a new malware is known, in order to have a fair comparision with these antiviruses, we need to consider new malwares. We use the sophisticated malware generator NGVCK available at VX Heavens [31] to generate 205 malwares. We obfuscate these malwares with self-modifying code, and we fed them to our tool and to well known antiviruses such as BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Baidu, Avast, and Symantec. Our tool was able to detect all these programs as malicious, whereas none of the well-known antiviruses was able to detect all these malwares. Table IV reports the detection rates of our tool and the well-known anti-viruses.

REFERENCES

[1] T.Touili and X.Ye, "Reachability analysis of self modifying code," in *ICECCS*, 2017.

[2] A. Bouajjani, J. Esparza, and O. Maler, "Reachability Analysis of Pushdown Automata: Application to Model Checking," in *CONCUR'97*, 1997.

[3] J.Esparza, D.Hansel, P.Rossmanith, and S.Schwoon, "Efficient algorithms for model checking pushdown systems," in *CAV*, 2000.

[4] J.Bergeron], M.Debbabi *et al.*, "Static detection of malicious code in executable programs," *Int. J. of Req. Eng*, vol. 2001, no. 184-189, 2001.

[5] G.Balakrishnan, T. Reps, N.Kidd, A.Lal, J.Lim *et al.*, "Model checking x86 executables with codesurfer/x86 and WPDS++," in *CAV*, 2005.

[6] P.Singh and A.Lakhotia, "Static verification of worm and virus behavior in binary executables using model checking," in *IAW*, 2003.

[7] J.Kinder, S.Katzenbeisser, C.Schallhart, and H.Veith, "Detecting malicious code by model checking," in *DIMVA*, 2005.

[8] F.Song and T.Touili, "Efficient malware detection using model-checking," in *FM*, 2012.

[9] P.Beaucamps, I.Gnaedig, and J.Marion, "Behavior abstraction in malware analysis," in *Runtime Verification*, 2010.

[10] F.Song and T.Touili, "Ltl model-checking for malware detection," in *TACAS*, 2013.

[11] H.Nguyen and T.Touili, "CARET model checking for malware detection," in *SPIN*, 2017.

[12] K.Dam and T.Touili, "Learning malware using generalized graph kernels," in *ARES*, 2018.

[13] ——, "Precise extraction of malicious behaviors," in *COMPSAC*, 2018.

[14] ——, "Malware detection based on graph classification," in *ICISSP*, 2017.

[15] H.Cai, Z.Shao, and A.Vaynberg, "Certified self-modifying code," *ACM SIGPLAN Notices*, vol. 42, no. 6, 2007.

[16] S.Debray, K.Coogan, and G.Townsend, "On the semantics of self-unpacking malware code," *Tech. rep. University of Arizona, Computer Science*, 2008.

[17] G.Bonfante, J.Marion, and D.Reynaud-Plantey, "A computability perspective on self-modifying programs," in *SEFM*, 2009.

[18] A.Bertrand, M.Matias, and D.Koen, "A model for self-modifying code," in *IHMMSec*, 2006.

[19] S.Blazy, V.Laporte, and D.Pichardie, "Verified abstract interpretation techniques for disassembling low-level self-modifying code," *JAR*, vol. 56, no. 3, 2016.

[20] K.Roundy and B.Miller, "Hybrid analysis and control of malware," in *RAID*, 2010.

[21] K.Coogan, S.Debray, T.Kaochar, and G.Townsend, "Automatic static unpacking of malware binaries," in *WCRE'09*, 2009.

[22] K.Gyung *et al.*, "Renovo: A hidden code extractor for packed executables," in *WORM*, 2007.

[23] P.Royal, M.Halpin *et al.*, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *ACSAC*, 2006.

[24] S.Schwoon, "Model-checking pushdown systems," Ph.D. dissertation, Technische Universität München, Universitätsbibliothek, 2002.

[25] H. J.Kinder, "Jakstab: A static analysis platform for binaries," in *CAV*, 2008.

[26] M.Vardi and P.Wolper, "Reasoning about infinite computations," *Inf. Comput.*, vol. 115, no. 1, 1994.

[27] P.Gastin and D.Oddoux, "Fast ltl to büchi automata translation," in *CAV*, 2001.

[28] U. Tool, "Automated unpacking: A behaviour based approach," https://github.com/malwaremusings/unpacker.

[29] VirusShare, "vxshare," https://virusshare.com.

[30] S.Cutler, "malshare," https://malshare.com.

[31] V.Heaven, "V.heavens," http://vxer.org/lib/.