CTL 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

***Abstract*—Self-modifying code is extensively used to obfuscate malware and to make reverse engineering harder. It consists in code that can modify its own instructions during the execution. Being able to analyse such code is crucial nowadays. In this paper, we consider the CTL model-checking problem of self modifying code. To model such programs, we use Self Modifying Pushdown Systems (SM-PDS), an extension of pushdown systems whose set of rules can be modified during execution. We reduce the CTL model-checking problem to the emptiness problem of Self-Modifying Alternating Büchi pushdown systems (SM-ABPDS). We implemented our techniques in a tool. We obtained encouraging results. In particular, our tool was able to detect several self-modifying malwares; it could even detect several malwares that well-known antiviruses such as McAfee, Norman, BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Avast, and Symantec failed to detect.**

I. Introduction

Self-modifying code is code that modifies its own instructions during its execution. It is usually used to enhance the difficulty of disassembling in reverse engineering. It is not only applied for software protection but also to make malwares harder to analyze and detect by static analysis approaches. Therefore, it is vital to analyze the programs equipped with self-modifying codes. In [1], [2], [3], [4], [5], [6], [7], [8], self-modifying codes implemented by packing and unpacking techniques have been well studied. Analysis tools for such codes are available [9], [10].

In this work, we consider another kind of implementation of self-modifying codes: **self-modifying instructions**. These are instructions that can read and write to memory, like the **mov** instruction that is able to copy data from one memory location to another. Using such self-modifying instructions can make malware detection harder. Fig. 1 shows how malware can use self-modifying instructions to evade from static analysis techniques. This figure shows a fragment of the malware Bagle.J equipped with such self-modifying instructions. First let us recall the semantics of the **mov** instruction. It copies the data item referred to by its second operand (register or memory location) into its first operand. In Fig. 1, in the box on the left, 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, `ff` is the binary code of the instruction `push`. Thus, the first line is translated to `push 0b`. The second instruction `mov 0x2 0xc` will replace the first byte at address `0x2` by `0xc`. Thus, at address `0x2`, `ff 0b` is replaced by `0c 0b`, i.e., the instruction `push 0b` is replaced by `jmp 0b`. If we analyse this code without taking into account the fact that `mov 0x2 0xc` is a self-modifying instruction, then, we will obtain the Control Flow Graph "CFG a", and we will reach the conclusion that the Bagle malicious behaviour implemented at address `0b` by the API functions RegCreateKeyA, RegDeleteValueA, and RegCloseKey is not reachable. However, the actual CFG is "CFG b", where the malicious fragment of the malware Bagle.J that starts at address `0b` is reached and will be executed.

It can be seen from this example that self-modifying codes can make malware detection harder, and that the **mov** instruction is able to modify instructions of the program successfully via its ability to read and write the memory. Thus, it is crucial to be able to analyse this kind of self-modifying code.

Self Modifying Pushdown Systems (SM-PDS) was proposed in [11] as a perfect model to represent such self-modifying codes. Indeed, a SM-PDS is an extension of standard Pushdown Systems (PDS) with self-modifying transition rules that modify the set of the rules of the PDS during the execution. This allows to represent the self-modifying instructions of the program. Moreover, SM-PDSs allow to keep track of the program's stack, which is very important for malware detection. Indeed, as decribed in [12], many obfuscation techniques rely on operations over the stack. Therefore, it is important for malware detection to have analysis techniques that can deal with the program stack.

[11] proposed efficient reachability analysis techniques for SM-PDSs. However, reachability is not enough to describe several malicious behaviors. For example, [13] showed that the branching-time temporal logic CTL is needed to represent several malicious behaviors. In this work, we go one step further and consider the CTL model-checking problem for SM-PDSs. This allows to detect CTL-like malicious behaviors on self-modifying code. We reduce this problem to the emptiness checking of Self-modifying Alternating Büchi Pushdown Systems (SM-ABPDS), and we propose an algorithm that computes a finite automaton that characterizes the set of configurations accepted by the SM-ABPDS. We implemented our techniques in a tool for self-modifying code analysis. We obtained encouraging results. Indeed, our tool was able to detect more than 700 self-modifying malwares. Amongst these malwares, several could not be detected by well known and widely used commercial anti-viruses such as McAfee, Norman, BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Avast, and Symantec.

Outline. The rest of the paper is organized as follows: In section 2, we recall the definition of SM-PDS. Section 3 reviews

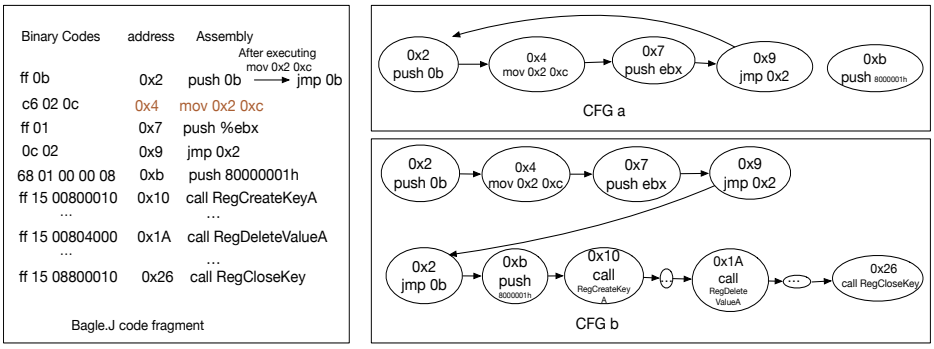


Fig. 1: An Example of Self-modifying Code

the definition of CTL and shows how CTL model checking on SM-PDSs can be reduced to the emptiness analysis on SM-ABPDSs. In Section 4, we give our algorithm that computes a finite automaton representing the set of configurations accepted by a SM-ABPDS. Our experiments are described in Sections 5. For lack of space, the proofs are ommitted. They can be found in the full version [14]. Moreover, the full version [14] contains also bigger experimental tables.

Related Work. Model checking and static analysis techniques were extensively applied for the analysis of binary code [1], [2], [3], [5], [15]. Malicious behaviors were described by temporal Logics in [16], [5], [15], [17], [18]. However, these works cannot deal with self-modifying code.

Formal semantics of self-modifying code was proposed in [19], [20], [21]. However, these specifications are too abstract to be used in practice. A new representation of self-modifying code called State Enhanced-Control Flow Graph (SE-CFG) was proposed in [22]. 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. Unlike SM-PDSs, this representation does not allow to take into account the stack of the program. Abstract interpretation techniques that compute an over-approximation of the set of reachable states of a self-modifying program were proposed in [23]. Combining static and dynamic analysis techniques to analyse self-modifying programs was applied in [24]. These techniques [23], [24] cannot handle the program's stack. Unpacking binary code is considered in [25], [26], [27], [21]. These works do not consider self-modifying **mov** instructions.

CTL model-checking for pushdown systems (PDS) was considered in [28], [13]. In [13], this problem is reduced to the emptiness problem in Alternating Büchi Pushdown Systems. In this work, we extend this approach to SM-PDSs. CTL model-checking for SM-PDSs can be reduced to CTL model-checking for PDSs. However, as witnessed by the results in the experiments section, this reduction is not efficient. We propose in this paper a *direct* and more efficient approach.

II. Self Modifying Pushdown Systems

In this section, we recall the definition of Self-modifying Pushdown Systems [11].

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 $\Delta_c \subseteq P \times (\Delta \cup \Delta_c) \times (\Delta \cup \Delta_c) \times P$ 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 $p \xrightarrow{(r_1,r_2)} p' \in \Delta_c$ 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 $p \in P$, $w \in \Gamma^*$ 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. A SM-PDS defines a transition relation $\Rightarrow_{\mathcal{P}}$ between configurations as follows: Let $p \in P, \gamma \in \Gamma, w \in \Gamma^*$, and $\theta \subseteq \Delta \cup \Delta_c$, then:

1) if $r \in \Delta_c$ is of the form $r = p \xrightarrow{(r_1,r_2)} p'$, 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 $(\langle p,\gamma w\rangle,\theta) \Rightarrow_{\mathcal{P}} (\langle p',w'w\rangle,\theta)$: 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 $\Rightarrow_{\mathcal{P}}$. A path of $\mathcal{P}$ is a sequence of configurations $c_0 c_1 ...$ s.t. for every $i \geq 0$,

$c_i \Rightarrow_{\mathcal{P}} c_{i+1}$. Then, for every $i \geq 0$, c_{i+1} is an immediate successor of c_i, and c_i is an immediate predecessor of c_{i+1}.

To simplify the presentation, we assume w.l.o.g. that $P = P_N \cup P_c$ s.t. $P_N \cap P_c = \emptyset$, and for every $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta$, $p \in P_N$ and for every $p \xhookrightarrow{(r_1,r_2)} p' \in \Delta_c$, $p \in P_c$ (we can always transform the rules of a given SM-PDS into equivalent ones that meet this condition).

SM-PDS vs. PDS. Let $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$ be a SM-PDS. It was shown in [11] that $\mathcal{P}$ can be translated to an equivalent pushdown system. The basic idea is to encode phases in the control points of the PDS (since the number of phases is finite). However, this translation is not efficient since the number of control points of the equivalent PDS is $|P| \cdot 2^{\mathcal{O}(|\Delta|+|\Delta_c|)}$.

From Self-modifying Codes to SM-PDS. SM-PDSs can be used to model self-modifying binary code as described in [11]. Our translation relies on the tool Jakstab [29] to disassemble binary code, determine indirect jumps, determine the program's instructions, and compute information about the values of the registers and the memory locations at each control point of the program. After getting the assembly program corresponding to the binary code using Jakstab [29], we translate it into a SM-PDS as described in [11]. In our translation, the control locations store the control points of the binary program and the stack mimics the program's stack. 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 [15] that models non self-modifying instructions of the program by a PDS. As for the self-modifying instructions of the program, they define the set of changing rules Δ_c. For more details, we refer the reader to [11].

III. CTL Model-Checking on SM-PDSs

A. The Computation Tree Logic CTL

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

$$\varphi ::= A | \neg A \mid \varphi \vee \varphi | \ \varphi \wedge \varphi \mid AX\varphi \mid EX\varphi \mid A[\varphi U \varphi] \mid E[\varphi U \varphi] \mid A[\varphi \widetilde{U} \varphi] \mid E[\varphi \widetilde{U} \varphi].$$

Given a CTL formula φ, the closure $cl(\varphi)$ is the set of all the subformulae of φ, including φ. Let $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$ be a SM-PDS, $\nu : P \to 2^{At}$ be a labelling function mapping to each control location $p \in P$ a set of atomic propositions. The satisfiability relation of a CTL formula φ at a configuration $(\langle p_0, w_0\rangle, \theta_0)$ (denoted by $(\langle p_0, w_0\rangle, \theta_0) \models_\nu \varphi$) is defined as follows:

- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu A$ iff $A \in \nu(p_0)$,
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu \neg A$ iff $A \notin \nu(p_0)$,
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu \varphi_1 \vee \varphi_2$ iff $(\langle p_0, w_0\rangle, \theta_0) \models_\nu \varphi_1$ or $(\langle p_0, w_0\rangle, \theta_0) \models_\nu \varphi_2$,
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu \varphi_1 \wedge \varphi_2$ iff $(\langle p_0, w_0\rangle, \theta_0) \models_\nu \varphi_1$ and $(\langle p_0, w_0\rangle, \theta_0) \models_\nu \varphi_2$,
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu AX\varphi$ iff $(\langle p_1, w_1\rangle, \theta_1) \models_\nu \varphi$ for every successor $(\langle p_1, w_1\rangle, \theta_1)$ of $(\langle p_0, w_0\rangle, \theta_0)$,
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu EX\varphi$ iff $(\langle p_1, w_1\rangle, \theta_1) \models_\nu \varphi$ for some successor $(\langle p_1, w_1\rangle, \theta_1)$ of $(\langle p_0, w_0\rangle, \theta_0)$,
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu A[\varphi_1 U \varphi_2]$ iff for every path $(\langle p_0, w_0\rangle, \theta_0)(\langle p_1, w_1\rangle, \theta_1)\cdots$ of $\mathcal{P}$ starting from $(\langle p_0, w_0\rangle, \theta_0)$, $\exists i \geq 0$ s.t. $(\langle p_i, w_i\rangle, \theta_i) \models_\nu \varphi_2$ and $\forall 0 \leq j < i, (\langle p_j, w_j\rangle, \theta_j) \models_\nu \varphi_1$.
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu E[\varphi_1 U \varphi_2]$ iff there exists a path $(\langle p_0, w_0\rangle, \theta_0)(\langle p_1, w_1\rangle, \theta_1)\cdots$ of $\mathcal{P}$ starting from $(\langle p_0, w_0\rangle, \theta_0)$, $\exists i \geq 0$ s.t. $(\langle p_i, w_i\rangle, \theta_i) \models_\nu \varphi_2$, $\forall 0 \leq j < i, (\langle p_j, w_j\rangle, \theta_j) \models_\nu \varphi_1$.
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu A[\varphi_1 \widetilde{U} \varphi_2]$ iff for every path $(\langle p_0, w_0\rangle, \theta_0)(\langle p_1, w_1\rangle, \theta_1)\cdots$ of $\mathcal{P}$ starting from $(\langle p_0, w_0\rangle, \theta_0)$, $\forall i \geq 0$, if $(\langle p_i, w_i\rangle, \theta_i) \not\models_\nu \varphi_2$, then $\exists 0 \leq j < i, (\langle p_j, w_j\rangle, \theta_j) \models_\nu \varphi_1$.
- $(\langle p_0, w_0\rangle, \theta_0) \models_\nu E[\varphi_1 \widetilde{U} \varphi_2]$ iff $\exists$ a path $(\langle p_0, w_0\rangle, \theta_0)(\langle p_1, w_1\rangle, \theta_1)\cdots$ of $\mathcal{P}$ starting with $(\langle p_0, w_0\rangle, \theta_0)$, s.t. $\forall i \geq 0$, if $(\langle p_i, w_i\rangle, \theta_i) \not\models_\nu \varphi_2$, then $\exists 0 \leq j < i, (\langle p_j, w_j\rangle, \theta_j) \models_\nu \varphi_1$.

Standard CTL operators can be expressed by the above operators: $\mathbf{EF}\psi = \mathbf{E}[true\mathbf{U}\psi]$, $\mathbf{AF}\psi = \mathbf{A}[true\mathbf{U}\psi]$, $\mathbf{EG}\psi = \mathbf{E}[false\widetilde{U}\psi]$, $\mathbf{AG}\psi = \mathbf{A}[false\widetilde{U}\psi]$.

B. Self-modifying Alternating Büchi Pushdown Systems

Definition 2. *A Self Modifying Alternating Büchi Pushdown System (SM-ABPDS) is a tuple $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, F)$, where P is a finite set of control points, Γ is a finite set of stack symbols, F is the set of final states, $\Delta \subseteq (P \times \Gamma) \times 2^{2^{\Delta \cup \Delta_c} \cup \{-\}} \times 2^{P \times \Gamma^*}$ is a finite set of transition rules in the form $\langle p, \gamma\rangle \xhookrightarrow{[\sigma_1, \cdots, \sigma_n]} \{\langle p_1, w_1\rangle, \cdots, \langle p_n, w_n\rangle\}$ where $[\sigma_1, \cdots, \sigma_n]$ is an ordered set and $\forall 1 \leq i \leq n, \sigma_i$ is either a set of rules $\sigma_i \subseteq \Delta \cup \Delta_c$ or $\sigma_i = -$, and $\Delta_c \subseteq P \times 2^{\Delta \cup \Delta_c} \times 2^{\Delta \cup \Delta_c} \times P$ is a finite set of modifying transition rules in the form $p \xhookrightarrow{(\sigma, \sigma')} p'$ where $\sigma, \sigma' \subseteq \Delta \cup \Delta_c$. A configuration of a SM-ABPDS is a tuple of the form $(\langle p, w\rangle, \theta)$ where $p \in P$, $w \in \Gamma^*$ and $\theta \subseteq \Delta \cup \Delta_c$ is the current phase.*

$\mathcal{BP}$ defines the transition relation $\Rightarrow_{\mathcal{BP}} \subseteq (P \times \Gamma^* \times 2^{\Delta \cup \Delta_c}) \times 2^{(P \times \Gamma^* \times 2^{\Delta \cup \Delta_c})}$ between configurations as follows: Let $\theta \subseteq \Delta \cup \Delta_c$, $\gamma \in \Gamma, w \in \Gamma^*$, and $p \in P$, then:

1) If $r : \langle p, \gamma\rangle \xhookrightarrow{[\sigma_1, \cdots, \sigma_m]} \{\langle p_1, w_1\rangle, \cdots, \langle p_m, w_m\rangle\}$ is a rule in $\Delta \cap \theta$, if either for every $1 \leq i \leq m, \sigma_i = -$ or $\exists 1 \leq i \leq m, \sigma_i \cap \theta \neq \emptyset$, then $(\langle p, \gamma w\rangle, \theta) \Rightarrow_{\mathcal{BP}} \{(\langle p_i, w_i w\rangle, \theta) | \sigma_i = -, 1 \leq i \leq m\} \cup \{(\langle p_i, w_i w\rangle, \theta) | \sigma_i \cap \theta \neq \emptyset, 1 \leq i \leq m\}$.
2) If $r : p \xhookrightarrow{(\sigma, \sigma')} p'$ is a rule in $\Delta_c \cap \theta$, then $(\langle p, w\rangle, \theta) \Rightarrow_{\mathcal{BP}} \{(\langle p', w\rangle, \theta')\}, \theta' = \theta \backslash \sigma \cup \sigma'$.

Intuitively, $[\sigma_1, \cdots, \sigma_m]$ in the transition $r : \langle p, \gamma\rangle \xhookrightarrow{[\sigma_1, \cdots, \sigma_m]} \{\langle p_1, w_1\rangle, \cdots, \langle p_m, w_m\rangle\}$ ensures that for a given configuration $(\langle p, \gamma w\rangle, \theta)$, for every $1 \leq i \leq n$, $(\langle p_i, w_i w\rangle, \theta)$ is in the set of immediate successor iff

- either for every $1 \leq j \leq n, \sigma_j = -$;
- or $\sigma_i = -$ and $\exists j \neq i, 1 \leq j \leq n$ s.t. $\sigma_j \cap \theta \neq \emptyset$
- or $\sigma_i \cap \theta \neq \emptyset$

Note that $-$ means that there is no constraint on whether θ contains a rule in σ_i or not.

For every $c \in P \times \Gamma^* \times 2^{\Delta \cup \Delta_c}$ and $C \subseteq P \times \Gamma^* \times 2^{\Delta \cup \Delta_c}$, if $c \Rightarrow_{\mathcal{BP}} C$ then c is an immediate predecessor of C and C is an immediate successor of c. Let $\Rightarrow^*_{\mathcal{BP}} \subseteq (P \times \Gamma^* \times 2^{\Delta \cup \Delta_c}) \times 2^{(P \times \Gamma^* \times 2^{\Delta \cup \Delta_c})}$ be the reflexive transitive closure of $\Rightarrow_{\mathcal{BP}}$ defined as follows: (1) $\forall c \in P \times \Gamma^* \times 2^{\Delta \cup \Delta_c}$, $c \Rightarrow^*_{\mathcal{BP}} \{c\}$, (2) if $c \Rightarrow_{\mathcal{BP}} C$, then $c \Rightarrow^*_{\mathcal{BP}} C$, and (3) if $c \Rightarrow_{\mathcal{BP}} \{c_1, ...c_n\}$ and $c_i \Rightarrow_{\mathcal{BP}} C_i$ for every $1 \leq i \leq n$, then $c \Rightarrow^*_{\mathcal{BP}} \bigcup_{i=1}^n C_i$. Given a set of configurations C, we define the sets $pre_{\mathcal{BP}}(C), pre^*_{\mathcal{BP}}(C)$ and $pre^+_{\mathcal{BP}}(C)$ as follows: $pre_{\mathcal{BP}}(C) = \{c \in P \times \Gamma^* \times 2^{\Delta \cup \Delta_c} \mid \exists C' \subseteq C$ s.t. C' is an immediate successor of $c\}$, $pre^*_{\mathcal{BP}}(C) = \{c \in P \times \Gamma^* \times 2^{\Delta \cup \Delta_c}, \exists C' \subseteq C$ s.t. $c \Rightarrow^*_{\mathcal{BP}} C'\}$ and $pre^+_{\mathcal{BP}}(C) = pre_{\mathcal{BP}} \circ pre^*_{\mathcal{BP}}(C)$. We omit the subscript $\mathcal{BP}$ when it is clear from the context.

A run ρ of $\mathcal{BP}$ starting from an initial configuration c_0 is a tree whose root is labelled by c_0 and whose other nodes are labelled by configurations of $P \times \Gamma^* \times 2^{\Delta \times \Delta_c}$. A node of ρ labelled by configuration c has n children labelled by $c_1, \ldots, c_n$, respectively, iff $c \Rightarrow_{\mathcal{BP}} \{c_1, ...c_n\}$. A path $c_0 c_1 \cdots$ of a run ρ is an infinite sequence of configurations s.t. c_0 is the root of ρ and c_{i+1} is one child of c_i. A path is accepting iff it visits some configurations with control locations in F infinitely often. A run is accepting iff all its paths are accepting. A configuration c is accepted by $\mathcal{BP}$ iff it is the root of a run accepted by $\mathcal{BP}$. The language of $\mathcal{BP}$, $L(\mathcal{BP})$, is the set of configurations accepted by $\mathcal{BP}$.

We assume w.l.o.g. that for every rule in Δ_c of the form $r : p \xrightarrow{(\sigma,\sigma')} p'$, $r \notin \sigma$.

Representing potentially infinite sets of configurations of SM-ABPDSs.

Alternating Multi-Automata (AMA) were introduced in [30] to finitely represent regular sets of configurations of an alternating PDS. In order to adapt AMA to represent regular sets of SM-ABPDS, we extend this notion taking phases into account as follows:

Definition 3. *Let $\mathcal{BP} = (P, \Gamma, \Delta, \Delta_c, F)$ be a SM-ABPDS. An Extended Alternating Multi-Automaton (EAMA) is a tuple $\mathcal{A} = (Q, \Gamma, T, I, Q_F)$ where $I \subseteq P \times 2^{\Delta \cup \Delta_c} \subseteq Q$ is the set of initial states, $T \subseteq Q \times (\Gamma \cup \{\epsilon\}) \times 2^Q$ is the set of transitions, $Q_F \subseteq Q$ is a finite set of final states.*

Let $\rightarrow_T$ be the transition relation defined as follows: (1) $\forall q \in Q, q \xrightarrow{\epsilon}_T \{q\}$ where ϵ is the empty word; (2) if $(q, \gamma, \{q_1, \cdots, q_n\}) \in T$, $q \xrightarrow{\gamma}_T \{q_1, \cdots, q_n\}$; and (3) if $q \xrightarrow{\gamma}_T \{q_1, \cdots, q_n\}$ and $q_i \xrightarrow{w}_T Q_i$ for every $1 \leq i \leq n$, then $q \xrightarrow{\gamma w}_T \bigcup_{i=1}^n Q_i$.

A configuration $(\langle p, w\rangle, \theta)$ is accepted by the EAMA $\mathcal{A}$ iff $(p, \theta) \in I$ and $\exists\, Q' \subseteq Q_F$ such that $(p, \theta) \xrightarrow{w}_T Q'$. Let $L(\mathcal{A})$ be the set of configurations accepted by $\mathcal{A}$. Let $\mathcal{C}$ be a set of configurations of the SM-ABPDS $\mathcal{BP}$. $\mathcal{C}$ is regular if there exists an EAMA $\mathcal{A}$ such that $\mathcal{C} = L(\mathcal{A})$.

C. From CTL Model-Checking of SM-PDSs to the emptiness problem of SM-ABPDSs

Let $\mathcal{P} = (P, \Gamma, \Delta, \Delta_c)$ be a Self Modifying Pushdown System with an initial configuration $c_0 = (\langle p_0, w_0\rangle, \theta_0)$. Given a set of atomic propositions At, let $\nu : P \rightarrow 2^{At}$ be a labeling function that associates each control location to a set of atomic propositions. Let φ be a CTL formula over At. Our goal is to check whether $c_0 \models_\nu \varphi$. This can be done by translating the SM-PDS into an equivalent PDS as described in Section II and in [11], and then applying the standard CTL model-checking algorithm for PDSs [31]. However, as will be shown in the experiments section (Section V), this approach is not efficient. Thus, we need a *direct* algorithm that operates directly on the SM-PDS without translating it into a PDS. We provide in this section a *direct* algorithm that performs CTL model-checking on SM-PDSs. To this aim, we will compute a kind of product of the SM-PDS with φ: we construct a Self Modifying Alternating Büchi Pushdown System $\mathcal{BP}_\varphi$ s.t. $\mathcal{BP}_\varphi$ accepts a configuration c iff $c \models_\nu \varphi$. Thus, determining whether $c_0 \models_\nu \varphi$ can be reduced to checking whether $c_0 \in L(\mathcal{BP}_\varphi)$.

Let $\mathcal{BP}_\varphi = (P', \Gamma, \Delta', \Delta'_c, F)$ be the SM-ABPDS defined as follows: $P' = P \times cl(\varphi) \cup P^{cl(\varphi)}$, where $P^{cl(\varphi)}$ is the set of control locations in the form p^ψ where $p \in P$ and $\psi \in cl(\varphi)$, $F = \{[p, a] \mid a \in cl(\varphi) \cap At$ and $a \in \nu(p)\} \cup \{[p, \neg a] \mid \neg a \in cl(\varphi), a \in At$ and $a \notin \nu(p)\} \cup P \times cl_{\widetilde{U}}(\varphi)$ where $cl_{\widetilde{U}}(\varphi)$ is the set of formulae of $cl(\varphi)$ in the form $E[\psi_1 \widetilde{U} \psi_2]$ or $A[\psi_1 \widetilde{U} \psi_2]$. In what follows, to compute Δ' and Δ'_c, every rule $r \in \Delta \cup \Delta_c$ leads to a set of rules $\{r'_1, \cdots, r'_n\}$ of $\Delta' \cup \Delta'_c$, we call this set of rules $prod(r)$. Moreover, let $prod_E(r) \subseteq prod(r)$ be the set of rules generated from r using subformulas of the form $EX\psi_1$, $E[\psi_1 U \psi_2]$ or $E[\psi_1 \widetilde{U} \psi_2]$(see below for more details about $prod(r)$ and $prod_E(r)$).

The transition relations Δ' and Δ'_c (resp. the sets $prod(r)$ and $prod_E(r)$, for every $r \in \Delta \cup \Delta_c$) are the smallest sets of transitions (resp. of sets of rules) defined as follows: Initially, $\Delta' = \Delta'_c = \emptyset$, $prod_E(r) = \emptyset$ and $prod(r) = \emptyset$, $\forall\, r \in \Delta \cup \Delta_c$. $\forall p \in P$, $\forall \psi \in cl(\varphi)$ and $\forall \gamma \in \Gamma$, we have:

1) if $\psi = a, a \in At$ and $a \in \nu(p)$; $\langle [p, a], \gamma\rangle \xrightarrow{[-]} \langle [p, a], \gamma\rangle \in \Delta'$
2) if $\psi = \neg a, a \in At$ and $a \notin \nu(p)$; $\langle [p, \psi], \gamma\rangle \xrightarrow{[-]} \langle [p, \psi], \gamma\rangle \in \Delta'$
3) if $\psi = \psi_1 \vee \psi_2$; $\langle [p, \psi], \gamma\rangle \xrightarrow{[-]} \langle [p, \psi_1], \gamma\rangle \in \Delta'$ and $\langle [p, \psi], \gamma\rangle \xrightarrow{[-]} \langle [p, \psi_2], \gamma\rangle \in \Delta'$
4) if $\psi = \psi_1 \wedge \psi_2$; $\langle [p, \psi], \gamma\rangle \xrightarrow{[-,-]} \{\langle [p, \psi_1], \gamma\rangle, \langle [p, \psi_2], \gamma\rangle\} \in \Delta'$
5) if $\psi = EX\psi_1$, then:
   a) if $p \in P_N$, for every $R = \langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta$, $R' = \langle [p, \psi], \gamma\rangle \xrightarrow{[-]} \langle [p', \psi_1], w\rangle \in \Delta'$, $R' \in prod_E(R)$ and $R' \in prod(R)$ ($R' \in prod(R)$ means that R' is generated from R and $R' \in prod_E(R)$ means that R' is generated from R using a formula of the form $EX\psi_1$, $E[\psi_1 U \psi_2]$ or $E[\psi_1 \widetilde{U} \psi_2]$.)
   b) if $p \in P_C$, for every $R = p \xrightarrow{(r_1, r_2)} p' \in \Delta_c$, $R' = [p, \psi] \xrightarrow{(\sigma, \sigma')} [p', \psi_1] \in \Delta'_c$ where $\sigma = prod(r_1), \sigma' = prod(r_2)$, $R' \in prod_E(R)$ and $R' \in prod(R)$
6) if $\psi = AX\psi_1$, then:

a) if $p \in P_N$, let $\{R_1 = \langle p,\gamma\rangle \hookrightarrow \langle p_1, w_1\rangle, \cdots, R_n = \langle p,\gamma\rangle \hookrightarrow \langle p_n, w_n\rangle\}$ be the set of all the rules of Δ that have $\langle p,\gamma\rangle$ in the left-hand-side. Then, $R' = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[\sigma_1,\cdots,\sigma_n]} \{\langle [p_1,\psi_1], w_1\rangle, \cdots, \langle [p_n,\psi_1], w_n\rangle\} \in \Delta'$, where for every $1 \le i \le n$, $\sigma_i = prod_E(R_i)$ and $R' \in prod(R_i)$.

b) if $p \in P_C$, let $\{R_1 = p \xhookrightarrow{(r_1,r_1')} p_1, \cdots, R_n = p \xhookrightarrow{(r_n,r_n')} p_n\}$ be the set of all the rules of Δ_c that have p in the left-hand-side. Then, for every $\gamma \in \Gamma$, $R'_\perp = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[\sigma_1,\cdots,\sigma_n]} \{\langle p_1^\psi,\gamma\rangle, \cdots, \langle p_n^\psi,\gamma\rangle\} \in \Delta'$ and for every $1 \le i \le n$, $R'_i : p_i^\psi \xhookrightarrow{(\sigma,\sigma')} [p_i,\psi_1] \in \Delta'_c$, where for every $1 \le i \le n, \sigma_i = prod_E(R_i)$, $\sigma = prod(r_i), \sigma' = prod(r_i')$, and for every $1 \le i \le n$, $R'_\perp, R' \in prod(R_i)$.

7) if $\psi = E[\psi_1 U \psi_2]$, then $\langle [p,\psi],\gamma\rangle \xhookrightarrow{[-]} \langle [p,\psi_2],\gamma\rangle \in \Delta'$ and:

a) if $p \in P_N$, for every $R = \langle p,\gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta$, $R' = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,-]} \{\langle [p,\psi_1],\gamma\rangle, \langle [p',\psi], w\rangle\} \in \Delta'$, $R' \in prod_E(R)$ and $R' \in prod(R)$.

b) if $p \in P_C$, for every $R = p \xhookrightarrow{(r_1,r_1')} p' \in \Delta_c$, then for every $\gamma \in \Gamma$, $R'_\perp = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,-]} \{\langle [p,\psi_1],\gamma\rangle, \langle p^\psi,\gamma\rangle\} \in \Delta'$ and $p^\psi \xhookrightarrow{(\sigma,\sigma')} [p',\psi] \in \Delta'_c$ where $\sigma = prod(r_1), \sigma' = prod(r_1')$, $R'_\perp, R' \in prod_E(R)$ and $R'_\perp, R' \in prod(R)$.

8) if $\psi = A[\psi_1 U \psi_2]$, then $\langle [p,\psi],\gamma\rangle \xhookrightarrow{[-]} \langle [p,\psi_2],\gamma\rangle \in \Delta'$, and:

a) if $p \in P_N$, let $\{R_1 = \langle p,\gamma\rangle \hookrightarrow \langle p_1, w_1\rangle, \cdots, R_n = \langle p,\gamma\rangle \hookrightarrow \langle p_n, w_n\rangle\}$ be the set of all the rules of Δ that have $\langle p,\gamma\rangle$ in the left-hand-side. Then, $R' = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,\sigma_1,\cdots,\sigma_n]} \{\langle [p,\psi_1],\gamma\rangle, \langle [p_1,\psi], w_1\rangle, \cdots, \langle [p_n,\psi], w_n\rangle\} \in \Delta'$ where for every $1 \le i \le n$, $\sigma_i = prod_E(R_i)$, and $R' \in prod(R_i)$.

b) if $p \in P_C$, let $\{R_1 = p \xhookrightarrow{(r_1,r_1')} p_1, \cdots, R_n = p \xhookrightarrow{(r_n,r_n')} p_n\}$ be the set of all the rules of Δ_c that have p in the left-hand-side. Then, $\forall 1 \le i \le n$, for every $\gamma \in \Gamma$, $R_\perp : \langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,\sigma_1,\cdots,\sigma_n]} \{\langle [p,\psi_1],\gamma\rangle, \langle p_1^\psi,\gamma\rangle, \cdots, \langle p_n^\psi,\gamma\rangle\} \in \Delta'$ and $R'_i : p_i^\psi \xhookrightarrow{(\sigma,\sigma')} [p_i,\psi] \in \Delta'_c$ where for every $1 \le i \le n, \sigma_i = prod_E(R_i)$, $\sigma = prod(r_i)$, $\sigma' = prod(r_i')$ and $R_\perp, R'_i \in prod(R_i)$.

9) if $\psi = E[\psi_1 \widetilde{U} \psi_2]$, then $\langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,-]} \{\langle [p,\psi_2],\gamma\rangle, \langle [p,\psi_1],\gamma\rangle\} \in \Delta'$ and:

a) if $p \in P_N$, then for every $R = \langle p,\gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta$, $R' = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,-]} \{\langle [p,\psi_2],\gamma\rangle, \langle [p',\psi], w\rangle\} \in \Delta'$, $R' \in prod_E(R)$ and $R' \in prod(R)$.

b) if $p \in P_C$, then for every $R = p \xhookrightarrow{(r_1,r_1')} p' \in \Delta_c$, for every $\gamma \in \Gamma$, $R_\perp = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,-]} \{\langle [p,\psi_2],\gamma\rangle, \langle p^\psi,\gamma\rangle\} \in \Delta'$ and $R' : p^\psi \xhookrightarrow{(\sigma,\sigma')} [p',\psi] \in \Delta'_c$ where $\sigma = prod(r_1), \sigma' = prod(r_1')$, $R_\perp, R' \in prod_E(R)$ and $R_\perp, R' \in prod(R)$.

10) if $\psi = A[\psi_1 \widetilde{U} \psi_2]$, then $\langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,-]} \{\langle [p,\psi_2],\gamma\rangle, \langle [p,\psi_1],\gamma\rangle\} \in \Delta'$, and:

a) if $p \in P_N$, let $\{R_1 = \langle p,\gamma\rangle \hookrightarrow \langle p_1, w_1\rangle, \cdots, R_n = \langle p,\gamma\rangle \hookrightarrow \langle p_n, w_n\rangle\}$ be the set of all the rules of Δ that have $\langle p,\gamma\rangle$ in the left-hand-side. Then for every $1 \le i \le n$, $\sigma_i = prod_E(R_i)$, $R' = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,\sigma_1,\cdots,\sigma_n]} \{\langle [p,\psi_2],\gamma\rangle, \langle [p_1,\psi], w_1\rangle, \cdots, \langle [p_n,\psi], w_n\rangle\} \in \Delta'$ and $R' \in prod(R_i)$.

b) if $p \in P_C$, let $\{R_1 = p \xhookrightarrow{(r_1,r_1')} p_1, \cdots, R_n = p \xhookrightarrow{(r_n,r_n')} p_n\}$ be the set of all the rules of Δ_c that have p in the left-hand-side. Then, for every $\gamma \in \Gamma, R_\perp = \langle [p,\psi],\gamma\rangle \xhookrightarrow{[-,\sigma_1,\cdots,\sigma_n]} \{\langle [p,\psi_2],\gamma\rangle, \langle p_1^\psi,\gamma\rangle, \cdots, \langle p_n^\psi,\gamma\rangle\} \in \Delta', \forall 1 \le i \le n, \sigma_i = prod_E(R_i)$ and for every $1 \le i \le n, R'_i : p_i^\psi \xhookrightarrow{(\sigma,\sigma')} [p_i,\psi] \in \Delta'_c$ where $\sigma = prod(r_i), \sigma' = prod(r_i')$ and $R_\perp, R'_i \in prod(R_i)$.

Let $prod(\Delta) = \{r' \in \Delta' \mid \exists r \in \Delta, r' \in prod(r)\}$ be the set of rules of Δ' that are generated from Δ. Let $\delta = \Delta' \backslash prod(\Delta)$ be the set of rules of Δ' that are not generated from any rule of Δ nor Δ_c (e.g., the rules computed by items 1, 2, 3 and 4 are in δ). These rules δ are independent of Δ and Δ_c. They are introduced by the structure of φ. Thus, they need to be present in all the phases of $\mathcal{BP}_\varphi$. Let then $\theta \subseteq \Delta \cup \Delta_c$ be a phase of $\mathcal{P}$. Its corresponding phase in $\mathcal{BP}_\varphi$ is $\beta(\theta) = prod(\theta) \cup \delta$, where $prod(\theta) = \{r' \in \Delta' \cup \Delta'_c \mid \exists r \in \theta, r' \in prod(r)\}$.

Let us explain the above construction intuitively. The above automaton $\mathcal{BP}_\varphi$ can be seen as a kind of product of the SM-PDS $\mathcal{P}$ with the formula φ. For $\psi \in cl(\varphi)$, $(\langle p, w\rangle, \theta) \models_\nu \psi$ iff $\mathcal{BP}_\varphi$ accepts a configuration $\big(\langle [p,\psi], w\rangle, \beta(\theta)\big)$. We give in what follows the intuition behind all the items above:

If $\psi = a \in At$, then $\forall w \in \Gamma^*, \theta \subseteq \Delta \cup \Delta_c$, $(\langle p, w\rangle, \theta) \models_\nu \psi$ iff $a \in \nu(p)$. Hence, the automaton $\mathcal{BP}_\varphi$ should accept a run starting from $(\langle [p,a], w\rangle, \beta(\theta))$ iff $a \in \nu(p)$. $[p,a] \in F$ iff $a \in \nu(p)$. Thus, the loop added in $(\langle [p,a], w\rangle, \beta(\theta))$ by Item 1 makes sure that $\mathcal{BP}_\varphi$ accepts this run.

If $\psi = \neg a$, then $\forall w \in \Gamma^*, \theta \subseteq \Delta \cup \Delta_c$, $(\langle p, w\rangle, \theta) \models_\nu \psi$ iff $a \notin \nu(p)$. Hence, the automaton $\mathcal{BP}_\varphi$ should accept a run starting from $(\langle [p,\neg a], w\rangle, \beta(\theta))$ iff $a \notin \nu(p)$. $[p,\neg a] \in F$ iff $a \notin \nu(p)$. Thus, the loop in $(\langle [p,\neg a], w\rangle, \beta(\theta))$ added by Item 2 ensures that $\mathcal{BP}_\varphi$ accepts this run.

If $\psi = \psi_1 \vee \psi_2$, then $\forall w \in \Gamma^*, \theta \subseteq \Delta \cup \Delta_c$, $(\langle p, w\rangle, \theta) \models_\nu \psi$ iff ($(\langle p, w\rangle, \theta) \models_\nu \psi_1$ or $(\langle p, w\rangle, \theta) \models_\nu \psi_2$). Thus, $\mathcal{BP}_\varphi$ accepts a run starting from $(\langle [p,\psi_1 \vee \psi_2], w\rangle, \beta(\theta))$ iff $\mathcal{BP}_\varphi$ has an accepting run starting from $(\langle [p,\psi_1], w\rangle, \beta(\theta))$ or $(\langle [p,\psi_2], w\rangle, \beta(\theta))$. This is ensured by Item 3. Item 4 is similar to Item 3, it handles the case $\psi = \psi_1 \wedge \psi_2$ where $(\langle p, w\rangle, \theta)$ satisfies ψ iff it satisfies both ψ_1 and ψ_2.

If $\psi = EX\psi_1$, then $\forall w \in \Gamma^*, \theta \subseteq \Delta \cup \Delta_c$, $(\langle p, w\rangle, \theta) \models_\nu \psi$ iff an immediate successor $(\langle p', w'\rangle, \theta')$ of $(\langle p, w\rangle, \theta)$ satisfies ψ_1. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $(\langle [p,\psi], w\rangle, \beta(\theta))$ iff it can accept a run from $(\langle [p',\psi_1], w'\rangle, \beta(\theta'))$. There are two cases depending on whether $p \in P_N$ or $p \in P_c$, because the form of the rules of the SM-PDS depends on whether $p \in P_N$ or $p \in P_c$: if $p \in P_N$, then necessarily, the rules that can be applied from p are of the form $\langle p,\gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta$, whereas if $p \in P_c$, then necessarily, the rules that can be applied from p are of the form $r : p \xhookrightarrow{(r_1,r_2)} p' \in \Delta_c$. Thus, if $p \in P_N$, then $\mathcal{BP}_\varphi$ has an accepting run from $(\langle [p,\psi],\gamma u\rangle, \beta(\theta))$ iff there exists a rule $\langle p,\gamma\rangle \hookrightarrow \langle p', w\rangle \in \Delta$ such that $\mathcal{BP}_\varphi$ has an accepting run from $(\langle [p',\psi_1], wu\rangle, \beta(\theta))$. This is ensured by Item 5(a). If $p \in P_c$, then $\mathcal{BP}_\varphi$ has an accepting run

from $(\langle[p,\psi],\gamma u\rangle,\beta(\theta))$ iff there exists a rule $r: p \xrightarrow{(r_1,r_2)} p' \in \Delta_c \cap \theta$ such that $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p',\psi_1],\gamma u\rangle,\beta(\theta'))$, where $\theta' = (\theta\backslash\{r_1\}) \cup \{r_2\}$. This is ensured by Item 5(b).

If $\psi = AX\psi_1$, then $\forall w \in \Gamma^*, \theta \subseteq \Delta \cup \Delta_c$, $(\langle p,w\rangle,\theta) \models_\nu \psi$ iff every immediate successor $(\langle p',w'\rangle,\theta')$ of $(\langle p,w\rangle,\theta)$ satisfies ψ_1. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p,\psi],w\rangle,\beta(\theta))$ iff it can accept a run from all its immediate successors $(\langle[p',\psi_1],w'\rangle,\beta(\theta'))$. As previously, there are two cases depending on whether $p \in P_N$ or $p \in P_c$: if $p \in P_N$, let $\gamma \in \Gamma$ and $u \in \Gamma^*$ such that $w = \gamma u$. Let then $\{\langle p,\gamma\rangle \hookrightarrow \langle p_1,w_1\rangle, \cdots, \langle p,\gamma\rangle \hookrightarrow \langle\ p_m,w_m\rangle\}$ be the set of all the rules of $\Delta \cap \theta$ that have $\langle p,\gamma\rangle$ in the left-hand-side. Then, $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p,\psi],\gamma u\rangle,\beta(\theta))$ iff $\mathcal{BP}_\varphi$ has an accepting run from every $(\langle[p_i,\psi_1],w_i u\rangle,\beta(\theta))$, $1 \le i \le m$. This is ensured by Item 6(a). Note that Item 6(a) considers all the rules $R_i : \langle p,\gamma\rangle \hookrightarrow \langle p_i,w_i\rangle$ that are in Δ (even those that are not in θ), then the constraints $[\sigma_1,\cdots,\sigma_n]$ of the rule R' of Item 6(a) ensures that only the R_i's that are in θ are applied. Note also that in $R', \sigma_i = prod_E(R_i)$ ensures that $\sigma_i \cap \beta(\theta) \neq \emptyset$ iff $R_i \cap \theta \neq \emptyset$. Here taking $\sigma_i = prod(R_i)$ is not correct because $R' \in prod(R_i)$ and so in this case, $\sigma_i \cap \beta(\theta)$ would always be nonempty. On the other hand, if $p \in P_c$, let $\{p \xrightarrow{(r_1,r_1')} p_1, \cdots, p \xrightarrow{(r_m,r_m')} p_m\}$ be the set of all the rules of $\Delta_c \cap \theta$ that have p in the left-hand-side. Then $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p,\psi],\gamma u\rangle,\beta(\theta))$ iff $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p_i,\psi_1],\gamma u\rangle,\beta(\theta_i))$, for every $1 \le i \le m$, where $\theta_i = (\theta\backslash\{r_i\}) \cup \{r_i'\}$. This is ensured by Item 6(b). As previously, Item 6(b) considers all the rules $R_i : p \xrightarrow{(r_i,r_i')} p_i$ that are in Δ_c (even those that are not in θ), then the constraints $[\sigma_1,\cdots,\sigma_n]$ of the rule $R'_\perp = \langle[p,\psi],\gamma\rangle \xrightarrow{[\sigma_1,\cdots,\sigma_n]} \{\langle p_1^\psi,\gamma\rangle,\cdots,\langle p_n^\psi,\gamma\rangle\}$ of Item 6(b) ensures that only the R_i's that are in θ are applied. Then $R_i' : p_i^\psi \xrightarrow{(\sigma,\sigma')} [p_i,\psi_1]$ ensures $\mathcal{BP}_\varphi$ has an accepting run from $(\langle p_i^\psi,\gamma u\rangle,\beta(\theta))$ iff $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p_i,\psi_1],\gamma u\rangle,\beta(\theta_i))$ where $\theta_i = (\theta\backslash\{r_i\}) \cup \{r_i'\}$ for $1 \le i \le n$. Note that $\sigma' = prod(r_i)$ and $\sigma' = prod(r_i')$, then $\beta(\theta_i) = \beta(\theta) \setminus \sigma \cup \sigma'$. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p,\psi],\gamma u\rangle,\beta(\theta))$ iff $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p_i,\psi_1],\gamma u\rangle,\beta(\theta_i))$ for every $1 \le i \le n$.

If $\psi = E[\psi_1 U \psi_2]$, then $\forall w \in \Gamma^*, \theta \subseteq \Delta \cup \Delta_c$, $(\langle p,w\rangle,\theta) \models_\nu \psi$ iff either it satisfies ψ_2 or it satisfies ψ_1 and there exists an immediate successor satisfying ψ. Thus, $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p,\psi],w\rangle,\beta(\theta))$ iff:

1) $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p,\psi_2],w\rangle,\beta(\theta))$. This is handled by the rules $\langle[p,\psi],\gamma\rangle \xrightarrow{[-,-]} \{\langle[p,\psi_2],\gamma\rangle,\langle[p,\psi_1],\gamma\rangle\}$ introduced by Item 7.
2) or $\mathcal{BP}_\varphi$ has an accepting run from both $(\langle[p,\psi_1],w\rangle,\beta(\theta))$ and $(\langle[p',\psi],w'\rangle,\beta(\theta'))$ where $(\langle p',w'\rangle,\theta')$ is an immediate successor of $(\langle p,w\rangle,\theta))$. There are two cases depending on whether $p \in P_N$ or $p \in P_C$: the case $p \in P_N$ is handled by Item 7(a). Its intuition is similar to the intuition behind the previous items. Let then $p \in P_C$. Then there exists a rule $r: p \xrightarrow{(r_1,r_1')} p' \in \theta \cap \Delta_c$ such that $\mathcal{BP}_\varphi$ has an accepting run from both $(\langle[p,\psi_1],w\rangle,\beta(\theta))$ and $(\langle[p',\psi],w\rangle,\beta(\theta'))$, where $\theta' = \theta\backslash\{r_1\} \cup \{r_1'\}$. This is ensured by the rule $R_\perp = \langle[p,\psi],\gamma\rangle \xrightarrow{[-,-]} \{\langle[p,\psi_2],\gamma\rangle,\langle p^\psi,\gamma\rangle\} \in \Delta'$ and $R' : p^\psi \xrightarrow{(\sigma,\sigma')} [p',\psi] \in \Delta_c'$ added by Item 7(b).

The case $\psi = A[\psi_1 U \psi_2]$ is handled in a similar way using Items 8. If $\psi = E[\psi_1 \widetilde{U} \psi_2]$, then $\forall w \in \Gamma^*, \theta \subseteq \Delta \cup \Delta_c$, $(\langle p,w\rangle,\theta) \models_\nu \psi$ iff it satisfies ψ_2 and either it satisfies also ψ_1, or it has a successor $(\langle p',w'\rangle,\theta')$ that satisfies ψ. Then, $\mathcal{BP}_\varphi$ has an accepting run from $(\langle[p,\psi],w\rangle,\theta)$ iff $\mathcal{BP}_\varphi$ has an accepting run from both $((\langle[p,\psi_2],w\rangle,\beta(\theta))$ and $(\langle[p,\psi_1],w\rangle,\beta(\theta)))$, or it has an accepting run from both $((\langle[p,\psi_2],w\rangle,\beta(\theta))$ and $(\langle[p',\psi],w'\rangle,\beta(\theta')))$. This case is handled by Items 9. To ensure that the runs on which ψ_2 always holds are accepted, we add $[p,\psi]$ in F. The case where $\psi = A[\psi_1 \widetilde{U} \psi_2]$ is handled similarly by Items 10.

We can show that (the proof is in the full version [14]):

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 $(\langle[p,\varphi],w\rangle,\beta(\theta))$.*

Therefore, CTL model-checking for SM-PDSs can be reduced to the problem of determining whether a SM-ABPDS has an accepting run (emptiness analysis).

IV. Computing the language of a SM-ABPDS

From now on, we fix a SM-ABPDS $\mathcal{BP} = (P,\Delta,\Delta_c,\Gamma,F)$. We show in this section that the set of configurations accepted by $\mathcal{BP}$ is regular and can be effectively represented by an EAMA (extended Alternating Multi-automaton). To this aim, we first characterize the set of configurations $L(\mathcal{BP})$ from which $\mathcal{BP}$ has an accepting run. Then we use this characterization to compute an EAMA that accepts it.

A. Characterizing $L(\mathcal{BP})$

Let $(X_i)_{i\ge 0}$ be the following sequence: $X_0 = P \times \Gamma^* \times 2^{\Delta\cup\Delta_c}$, and for every $i \ge 0$, $X_{i+1} = Pre^+_{\mathcal{BP}}\big(X_i \cap (F \times \Gamma^* \times 2^{\Delta\cup\Delta_c})\big)$. Let $Y_{\mathcal{BP}} = \bigcap_{i\ge 0} X_i$. We can show that $L(\mathcal{BP}) = Y_{\mathcal{BP}}$:

Theorem 2. *A SM-ABPDS $\mathcal{BP}$ has an accepting run starting from a configuration $(\langle p,w\rangle,\theta)$ iff $(\langle p,w\rangle,\theta) \in Y_{\mathcal{BP}}$.*

Proving this theorem is based on the following lemma:

Lemma 1. *$\mathcal{BP}$ has a run ρ starting from a configuration $(\langle p,w\rangle,\theta)$ s.t. each path of ρ visits configurations with control locations in F at least k times iff $(\langle p,w\rangle,\theta) \in X_k$.*

Indeed, let $c \in X_1$. Then c has a successor $C \subseteq F \times \Gamma^* \times 2^{\Delta\cup\Delta_c}$ (since $X_1 = Pre^+_{\mathcal{BP}}(X_0 \cap (F \times \Gamma^* \times 2^{\Delta\cup\Delta_c}))$). Therefore, $\mathcal{BP}$ has a run starting from c that visits some configuration $p \in F$ at least once. $X_2 = Pre^+_{\mathcal{BP}}\big(X_1 \cap (F \times \Gamma^* \times 2^{\Delta\cup\Delta_c})\big)$, thus $\forall c' \in X_2$, a run starting from c' will visit configurations in $X_1 \cap (F \times \Gamma^* \times 2^{\Delta\cup\Delta_c})$ at least once; and thus, it visits configurations with control locations in F at least

twice. Thus, we can get by induction that $\forall k \geq 1$, for every configuration c in X_k, $\mathcal{BP}$ has a run that visits configurations with control locations in F at least k times.

B. Computing $Y_{\mathcal{BP}}$

In this section, our goal is to compute $Y_{\mathcal{BP}}$. We show that this set can be effectively represented by an EAMA $\mathcal{A} = (Q, \Gamma, T, I, Q_f)$, where $Q \subseteq P \times 2^{\Delta \cup \Delta_c} \times \mathbb{N} \cup \{q_f\}$, $I \subseteq P \times 2^{\Delta \cup \Delta_c} \times \mathbb{N}$ is the set of initial states and q_f is the final state ($Q_f = \{q_f\}$). Following [13], we propose a saturation procedure to compute $\mathcal{A}$ iteratively. **Algorithm 1** below computes $\mathcal{A}$. Intuitively, it computes the different X_i's iteratively. Each iteration step i computes an EAMA $\mathcal{A}_i$. States of $\mathcal{A}_i$ are of the form $(p, \theta)^i$, where $p \in P$ and $\theta \subseteq \Delta \cup \Delta_c$. There are two loops in the algorithm: the outer loop ($loop_1$) and the inner loop ($loop_2$). As will be explained later, if the sequence (X_i) is strictly decreasing, the outer loop won't terminate. So we introduce two projections to force termination as follows: for every $S \subseteq P \times 2^{\Delta \cup \Delta_c} \times \mathbb{N} \cup \{q_f\}$:

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

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

1: **Initially:** $i = 0, T = \{(q_f, \gamma, \{q_f\})\}, \forall \gamma \in \Gamma, p \in P, \theta \subseteq \Delta \cup \Delta_c, (p, \theta)^0 = q_f$.
2: **Repeat** (we call this loop $loop_1$)
3: $i := i + 1$;
4: $\forall (p, \theta)^{i-1}$ in the current automaton s.t. $p \in F$, add $(p, \theta)^i \xrightarrow{\epsilon} (p, \theta)^{i-1}$ to T
5: **Repeat** (we call this loop $loop_2$)
6: **if** $r : \langle p, \gamma \rangle \xrightarrow{[\sigma_1, \cdots, \sigma_n]} \{\langle p_1, w_1 \rangle, \cdots, \langle p_n, w_n \rangle\} \in \Delta \cap \theta$
7: **if** $(\exists k, 1 \leq k \leq n : \sigma_k \cap \theta \neq \emptyset$ or $\forall k, 1 \leq k \leq n, \sigma_k = -)$
8: Add $(p, \theta)^i \xrightarrow{\gamma} S_Q$ to T, where $S_Q = \{q \in Q_k | (p_k, \theta)^i \xrightarrow{w_k}_T Q_k, 1 \leq k \leq n$ s.t. $\sigma_k = -$ or $\sigma_k \cap \theta \neq \emptyset\}$.
9: **if** $r : p \xrightarrow{(\sigma, \sigma')} p' \in \Delta_c \cap \theta$, s.t.
10: $(p', \theta')^i \xrightarrow{\gamma}_T Q$ and $\theta' = \theta \backslash \sigma \cup \sigma'$
11: Then, add $(p, \theta)^i \xrightarrow{\gamma} Q$ to T.
12: **Until** No new transition rule can be added.
13: Remove from T the transition rules $(p, \theta)^i \xrightarrow{\epsilon} (p, \theta)^{i-1}$, for every $p \in F$.
14: Replace every $(p, \theta)^i \xrightarrow{\gamma} S$ in T by $(p, \theta)^i \xrightarrow{\gamma} \pi^i(S), \forall p \in P, \gamma \in \Gamma, S \subseteq Q$
15: **Until** $i > 1$ and for every $p \in P, \gamma \in \Gamma, \theta \in 2^{\Delta \cup \Delta_c}, S \subseteq P \times 2^{\Delta \cup \Delta_c} \times \{i\} \cup \{q_f\}, (p, \theta)^i \xrightarrow{\gamma} S \in T \iff (p, \theta)^{i-1} \xrightarrow{\gamma} \pi^{-1}(S) \in T$.

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

Intuitively, at each step i, every state (p, θ) is represented by state $(p, \theta)^i$ in $\mathcal{A}_i$. For every $(p, \theta) \in I$, $\mathcal{A}_i$ recognizes a configuration $(\langle p, w \rangle, \theta)$ if $(p, \theta)^i \xrightarrow{\omega}_T q_f$. $\mathcal{A}_0$ is the automaton obtained by **Line 1**. It accepts $X_0 = P \times \Gamma^* \times 2^{\Delta \cup \Delta_c}$. At the beginning of each iteration, an ϵ-transition in the form $(p, \theta)^i \xrightarrow{\epsilon}_T (p, \theta)^{i-1}$ is added in **Line 4** for every $(p, \theta) \in F \times 2^{\Delta \cup \Delta_c}$ in the current automaton. This allows to get $L(\mathcal{A}_{i-1}) \cap (F \times \Gamma^* \times 2^{\Delta \cup \Delta_c})$. **Lines 5-12** ($loop_2$) is the saturation procedure that computes $Pre^*_{\mathcal{BP}}\big(L(\mathcal{A}_{i-1}) \cap (F \times \Gamma^* \times 2^{\Delta \cup \Delta_c})\big)$. They ensure that if θ is a phase such that

$$\langle p, \gamma \rangle \xrightarrow{[\sigma_1, \cdots, \sigma_n]} \{\langle p_1, w_1 \rangle, \cdots, \langle p_n, w_n \rangle\} \in \Delta \cap \theta$$

, s.t. either $\exists k, 1 \leq k \leq n, \sigma_k \cap \theta \neq \emptyset$ or $\forall k, 1 \leq k \leq n, \sigma_k = -$, and for every k s.t. $\sigma_k \cap \theta \neq \emptyset$ or $\sigma_k = -$, $(\langle p_k, w_k w \rangle, \theta) \in L(\mathcal{A}_i)$ (i.e., $(p_k, \theta)^i \xrightarrow{w_k w}_T q_f$), then $(\langle p, \gamma w \rangle, \theta)$ should also be in $L(\mathcal{A}_i)$ (since it is a predecessor of $\{(\langle p_k, w_k w \rangle, \theta), 1 \leq k \leq n\}$). I.e., T should contain the path $(p, \theta)^i \xrightarrow{\gamma w}_T q_f$. This path is added thanks to Line 8. Moreover, if θ is a phase such that $\langle p, \gamma \rangle \xrightarrow{(\sigma, \sigma')} p' \in \Delta_c \cap \theta$ and $(\langle p', \gamma w \rangle, \theta') \in L(\mathcal{A}_i)$ (i.e., $(p', \theta')^i \xrightarrow{\gamma w}_T q_f$), where $\theta' = \theta \backslash \sigma \cup \sigma'$; then $(\langle p, \gamma w \rangle, \theta)$ should also be in $L(\mathcal{A}_i)$ (since it is a predecessor of $(\langle p', \gamma w \rangle, \theta')$). I.e., T should contain the path $(p, \theta)^i \xrightarrow{\gamma w}_T q_f$. This path is added thanks to Line 11. **Line 13** removes the ϵ-transition added by Line 4. This leads to $Pre^+_{\mathcal{BP}}\big(L(\mathcal{A}_{i-1}) \cap (F \times \Gamma^* \times 2^{\Delta \cup \Delta_c})\big)$.

Let **Algorithm A** be **Algorithm 1** without Line 14. Then, if **Algorithm A** terminates, it computes $Y_{\mathcal{BP}}$. However, if the sequence X_i is strictly decreasing, **Algorithm A** never terminates. **Lines 14-15** are then used to force termination. Indeed, thanks to the substitution of **Line 14**, at the end of step i, states of the form $(p, \theta)^j$, for $j < i$ become useless and can be removed. **Line 15** checks then whether at step i, the transitions of $\mathcal{A}_i$ are "the same" than those of $\mathcal{A}_{i-1}$. If this is the case, the algorithm terminates. Termination of the algorithm then follows from the fact that step i adds less transitions than step $i-1$. Intuitively, this is due to the fact that $L(\mathcal{A}_i) \subseteq L(\mathcal{A}_{i-1})$, because step i computes $Pre^+_{\mathcal{BP}}\big(L(\mathcal{A}_{i-1}) \cap (F \times \Gamma^* \times 2^{\Delta \cup \Delta_c})\big)$ and $\mathcal{A}_0$ accepts $P \times \Gamma^* \times 2^{\Delta \cup \Delta_c}$.

Thus, we can show that (a detailed proof can be found in the full version [14]):

Theorem 3. ***Algorithm 1*** *always terminates and produces* $Y_{\mathcal{BP}}$.

Thus, it follows from Theorems 1, 2 and 3 that:

Corollary 1. *Let $\mathcal{P}$ be a SM-PDS, $\nu : P \to 2^{At}$ be a labelling function, and φ be a CTL formula over At. Then, we can compute an EAMA $\mathcal{A}$ that characterizes the set of configurations $(\langle p, w \rangle, \theta)$ of $\mathcal{P}$ such that $(\langle p, w \rangle, \theta) \models_\nu \varphi$.*

V. Experiments

A. Our algorithm vs. standard CTL on PDSs

We implemented our algorithm in a tool and we compared its performance with the approach that consists in translating the SM-PDS to an equivalent standard PDS, and then applying the standard CTL model checking algorithm implemented in the PDS model-checker tool PuMoC [31]. All our experiments

$\|\Delta\| + \|\Delta_c\|$	formula size	SM-PDS time (s)	To PDS	PDS	Total Time	$Result1$	$Result2$
$6+4$	5	**0.36s**	0.21s	0.45s	0.66s	Y	Y
$8+4$	12	**2.88s**	0.35s	3.41s	3.76s	Y	Y
$20+4$	15	**3.84s**	0.62s	3.94s	4.56s	N	N
$500+8$	6	**20.51s**	17.02s	29.25s	46.27s	N	N
$600+9$	8	**23.34s**	295.24s	57.79s	353.03s	Y	Y
$1000+10$	6	**35.11s**	3251.02s	7127.41s	10378.43s	N	N
$1100+10$	45	**83.63s**	3251.02s	-	-	N	-
$1500+8$	30	**60.71s**	2182.78s	13821.34s	16004.12s	N	N
$2000+10$	18	**49.48s**	5529.30s	-	-	Y	-
$2000+10$	36	**61.13s**	5529.30s	-	-	N	-
$2100+10$	15	**60.74s**	5544.69s	-	-	Y	-
$3800+10$	30	**99.06s**	9295.24s	-	-	N	-
$3850+10$	8	**93.20s**	9308.01s	-	-	Y	-
$3850+10$	30	**115.52s**	9308.01s	-	-	N	-
$4800+11$	10	**142.13s**	42184.85s	-	-	Y	-
$4800+11$	15	**153.22s**	42184.85s	-	-	Y	-
$5500+10$	20	**196.46s**	45745.44s	-	-	Y	-
$10180+16$	5	**782.91s**	2134643.52s	-	-	N	-
$10180+16$	8	**1041.87s**	2134643.52s	-	-	N	-
$10180+16$	10	**1129.36s**	2134643.52s	-	-	Y	-

TABLE I: Our approach vs. standard CTL model checking for PDSs

were run on Ubuntu 16.04 with a 2.7 GHz CPU, 2GB of memory. To perform this comparision, we randomly generate several SM-PDSs and CTL formulas. Our results (CPU Execution time) are shown in Table I. **Column** $|\Delta| + |\Delta_c|$ indicates the size of the transition rules. **Column** formula size shows the size of the CTL formula. **Column** SM-PDS is the cost of our direct algorithm. **Column** To PDS reports the cost it takes to get the equivalent PDS from the SM-PDS. **Column** PDS is the cost used to run standard CTL model checking for the equivalent PDS in PuMoC. **Column** Total Time is the whole cost it takes to translate the SM-PDS into a PDS, and then apply the PDS CTL model-checking algorithm of PuMoC [31] (Total Time= To PDS + PDS). **Column** $Result1$ is the result of our approach and $Result2$ is the result of PuMoC [31], where Y means yes the formula is satisfied and N means no, the formula is not satisfied. "-" means out of memory. It can be seen that our direct approach is much more efficient, and that it terminates in all the cases, whereas going through CTL model-checking of PDSs gets out of memory in most of the cases. **Translating the SM-PDS to a standard PDS may take more than 24 days, whereas our direct algorithm takes only a few seconds.** More results can be found in the full version [14].

B. Malicious Behavior Detection on Self-Modifying Code

1) Specifying malicious behaviors using CTL.: We applied our tool to detect several self-modifying malwares. Indeed, as shown in [32], several malicious behaviors can be described by CTL formulas. We give in what follows two examples of such malicious behaviors. More examples can be found in the full version [14].

Appending Viruses. 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 CTL formula:

$$\phi_{av} = \bigvee \mathbf{EF}\Big(call \wedge \mathbf{EX}(\text{top-of-stack} = a) \wedge \mathbf{AG}\neg\big(ret \wedge (\text{top-of-stack} = a)\big)\Big)$$

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 $call \wedge \mathbf{EX}(\text{top-of-stack} = a)$ 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: 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.

Spyware (Scanning the Disk). The aim of a spyware is to steal information from the host. To do this, it has to scan the disk of the host in order to find the interesting file that he wants to steal. If a file is found, it will run a payload to steal it, then continues searching the next file. If a directory is found, it will enter this path and continues scanning. This malicious behaviour is present e.g. in the notorious spyware Flame: It first calls the function FindFirstFileW to search the first object in the given path, then, it will check whether the function call succeeds or not. If the function call fails, it will call the function GetLastError. Otherwise it will call either the function FindFirstFileW again if it finds a directory or the

Example	Size	Result	*cost*	Example	Size	Result	*cost*	Example	Size	Result	*cost*
calculation.exe	9952	No	76.34s	cisvc.exe	4105	No	31.22s	simple.exe	52	No	3.17s
shutdown.exe	2529	No	23.52s	loop.exe	529	No	11.78s	cmd.exe	1324	No	19.36s
notepad.exe	10529	No	68.77s	java.exe	800	No	19.17s	java.exe	21324	No	122.07s
sort.exe	8529	No	74.12s	bibDesk.exe	32800	No	243.79s	interface.exe	1005	No	18.25s
ipv4.exe	968	No	24.43s	TextWrangler.exe	14675	No	65.09s	sogou.exe	45219	No	301.14s
game.exe	34325	No	234.14s	cycle.tex	9014	No	75.44s	calender.exe	892	No	25.39s
Adson.1651	39	Yes	0.44s	Adson.1734	42	Yes	0.43s	Alcaul.d	40	Yes	0.48s
Adon.1703	37	Yes	0.39s	Adon.1559	37	Ye s	0.35s	Alcaul.i	48	Yes	0.44s
Alcaul.o	33	Yes	0.29s	Alcaul.d	845	Yes	0.165s	Alaul.c	355	Yes	0.109s
Alcaul.j	45	Yes	0.56s	Alcaul.m	23	Yes	0.19s	Evol.a	53	Yes	7.09s
Alcaul.e	32	Yes	1.93s	Alcaul.h	34	Yes	3.95s	Alcaul.g	25	Yes	4.18s
Alcaul.b	19	Yes	0.12s	Alcaul.f	23	Yes	1.99s	Alcaul.k	28	Yes	2.31s
Alcaul.l	27	Yes	0.95s	Klinge	45	Yes	64.15s	Akez.Win32.5	490	Yes	53.18s
Netsky.a	45	Yes	19.12s	Mydoom.c	155	Yes	4.14s	MyDoom-N	16980	Yes	343.93s
Netsky.x	55	Yes	21.85s	Netsky.y	68	Yes	29.06s	Netsky.z	115	Yes	43.37s
Netsky.gen	5508	Yes	59.24s	Netsky.p	6015	Yes	76.32s	Netsky.m	6805	Yes	73.77s
Netsky.r	230	Yes	8.83s	Netsky.k	6115	Yes	79.79s	Netsky.e	6245	Yes	79.44s
Mydoom.y	26902	Yes	452.77s	Mydoom.j	22355	Yes	211.93s	klez-N	6281	Yes	63.07s
klez.c	30	Yes	2.79s	Mydoom.v	5965	Yes	283.11s	Netsky.b	45	Yes	29.51s
LdPinch.dd	8230	Yes	48.17s	LdPinch.gw	2050	Yes	39.92s	LdPinch.dh	3744	Yes	41.01s
LdPinch.c	7363	Yes	52.46s	LdPinch.ck	8228	Yes	49.77s	LdPinch.gen	7245	Yes	62.19s
LdPinch.ld	6631	Yes	29.70s	LdPinch.er	3910	Yes	32.73s	LdPinch.cf	4366	Yes	27.26s
Muma.c	15915	Yes	114.47s	LdPinch.au	8245	Yes	92.24s	LdPinch.fi	3138	Yes	74.39s
Gismor	1140	Yes	23.56s	Botter.a	5030	Yes	71.42s	Navidad.a	3750	Yes	46.59s
Bagle.ab	5690	Yes	89.42s	Bagle.ef	995	Yes	54.11s	Bagle.eg	380	Yes	25.49s
Atak.f	2005	Yes	11.35s	Atak.g	2498	Yes	16.69s	Atak.l	1914	Yes	10.37s
Newapt.C	11730	Yes	924.92s	Krynos.b	18370	Yes	893.45s	Jeans.a	6490	Yes	188.36s
Bagle.m	5111	Yes	39.92s	Bagle.k	35	Yes	1.92s	Bagle.t	3345	Yes	45.64s
LdPinch.aaz	4145	Yes	41.05s	LdPinch.c0	8230	Yes	65.17s	LdPinch.ee	6501	Yes	71.30s
LdPinch.v	7235	Yes	51.69s	LdPinch.fk	4906	Yes	47.11s	LdPinch.awp	195	Yes	17.97s
LdPinch.bb	8145	Yes	63.13s	LdPinch.br	3645	Yes	33.52s	LdPinch.hb	1645	Yes	21.08s
LdPinch.by	970	Yes	42.92s	Generic.2026199	433	Yes	32.83s	LdPinch.arr	1250	Yes	49.84s
Mydoom.M	5965	Yes	75.19s	MyDoom.54464	5935	Yes	45.78s	Mydoom.e	138	Yes	46.53s
Mydoom.ACQ	19210	Yes	439.57s	Mydoom.ba	19423	Yes	238.77s	Mydoom.ftde	19495	Yes	339.29s
Sramota.avf	240	Yes	11.01s	Mydoom	238	Yes	2.01s	Mydoom.288	248	Yes	3.12s
Mydoom.R	230	Yes	30.22s	Mydoom.dlnpqi	235	Yes	1.99s	Mydoom.o	235	Yes	2.01s
klez.e	27	Yes	3.94s	Magistr.b	4670	Yes	231.97s	Magistr.a.poly	36989	Yes	469.63s
Kelino.g	470	Yes	22.08s	Plage.b	395	Yes	1.96s	Urbe.a	123	Yes	9.17s
Kelino.l	495	Yes	21.01s	Kipis.t	20378	Yes	121.11s	klez.d	31	Yes	0.95s
Netsky.d	45	Yes	1.87s	Ardurk.d	1913	Yes	12.08s	klez.f	27	Yes	0.73s
Repah.b	221	Yes	12.76s	Gibe.b	5358	Yes	37.01s	Magistr.b	4670	Yes	43.59s
NGVCK1	329	Yes	2.33s	NGVCK2	455	Yes	2.89s	NGVCK3	2300	Yes	111.20s
NGVCK4	550	Yes	9.19s	NGVCK5	1555	Yes	10.25s	NGVCK6	1698	Yes	21.07s
NGVCK7	6902	Yes	14.24s	NGVCK8	2355	Yes	54.76s	NGVCK9	281	Yes	12.31s
NGVCK10	2980	Yes	23.51s	NGVCK11	5965	Yes	51.92s	NGVCK12	4529	Yes	36.14s
NGVCK13	2210	Yes	18.12s	NGVCK14	5358	Yes	120.04s	NGVCK15	970	Yes	42.12s
NGVCK16	658	Yes	6.59s	NGVCK17	913	Yes	2.03s	NGVCK18	90	Yes	0.89s
NGVCK19	1295	Yes	16.58s	NGVCK20	4378	Yes	32.41s	NGVCK21	31	Yes	0.77s
NGVCK22	370	Yes	2.08s	NGVCK23	3955	Yes	36.74s	NGVCK24	6924	Yes	92.03s

TABLE II: Partial Experimental Results

our tool	McAfee	Norman	BitDefender	Kinsoft	Avira	eScan	Kaspersky	Qihoo360	Avast	Symantec
100%	38.0%	33.3%	48.5%	31.4%	11.6%	21.9%	70.4 %	1.4%	2.3%	57.1%

TABLE III: Our tool v.s. well known anti-viruses

function FindNextFileW to search for the next object. We can specify this behavior in CTL as follows:

$$\phi_{spy} = \mathbf{EF}\left(call\ FindFirstFileW \wedge \mathbf{AF}\left(call\ GetLastError \vee call\ FindFirstFileW \vee call\ FindNextFileW\right)\right)$$

This formula states that there exists a path where the function $FindFirstFileW$ is called, then, in all the future paths, the program either calls $GetLastError$ (if $FindFirstFileW$ failed) or calls $FindFirstFileW$ (if a directory is found) or calls $FindNextFileW$ (to search for the next file). Scanning a disk can be a behavior of a benign program. To avoid false alarms, we can combine this CTL formula with other formulas describing other malicious behaviors expressing the payload (such as sending a file) to determine whether the binary code is a malware or not. Note that, the formula is branching time and cannot be described as a LTL formula.

2) Applying our tool for malware detection.: We applied our tool to detect several malwares. We use Jakstab [29] as disassembler. We consider 400 email-worms, 30 worms and 100 viruses from VX heaven[33] and 260 new malwares generated by NGVCK, one of the best malware generators. We also

choose 19 benign samples from Windows XP system (win32). 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. First, we abstract away the semantics of the self-modifying instructions and model such programs as standard PDSs as described in [32], and perform CTL model-checking for PDSs to determine whether the programs contain any malicious behavior. In this case, *none* of the programs was declared as malicious. Then, we use SM-PDSs to model these programs, thus, taking self-modifying instructions into consideration. Then, we check whether these SM-PDSs satisfy any malicious CTL formula in our database (CTL formulas described above and in the full version). If yes, the program is declared as malicious. If not, it is declared as benign. In our experiments (we have 790 malwares), our tool was able to detect all these programs as malicious (whereas when we model these programs using standard PDSs and abstract away self-modifying instructions, none of these programs was detected as malicious). Our tool was also able to determine that benign programs are benign. We report in Table II some of the results we obtained. More results can be found in the full version. **Column** Size gives the number of control locations, **Column** Result shows the result of our algorithm: Y means malicious and N means benign; and **Column** cost gives the cost in seconds. You can see that our CTL model checking approach allows to detect all the malicious programs in a few seconds.

3) Comparision with well-known antiviruses.: We compare our tool against well-known and widely used antiviruses such as McAfee, Norman, BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Avast, and Symantec. To have a fair comparision, we need to consider unknown malwares. Thus, we generated 105 malwares that include self-modifying codes. We use the sophisticated malware generator NGVCK available at VX Heavens [33] to obtain malicious codes and we obfuscate them using self-modifying instructions. 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 III shows the detection rates of our tool v.s. the well-known anti-viruses.

REFERENCES

[1] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, N. Tawbi *et al.*, "Static detection of malicious code in executable programs," *Int. J. of Req. Eng*, vol. 2001, no. 184-189, p. 79, 2001.

[2] G. Balakrishnan, T. W. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. H. Yong, C. Chen, and T. Teitelbaum, "Model checking x86 executables with codesurfer/x86 and WPDS++," in *CAV*, 2005, pp. 158–163.

[3] P. K. Singh and A. Lakhotia, "Static verification of worm and virus behavior in binary executables using model checking," in *IAW*, 2003, pp. 298–300.

[4] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2003.

[5] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *DIMVA*, 2005, pp. 174–187.

[6] F. Song and T. Touili, "Pushdown model checking for malware detection," *STTT*, vol. 16, no. 2, pp. 147–173, 2014.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *S&P*, 2005, pp. 32–46.

[8] G. Bonfante, H. Godfroy, and J.-Y. Marion, "A construction of a self-modifiying language with a formal correction proof," in *MALWARE*. IEEE, 2017, pp. 99–106.

[9] E. R. Team, "Self-modifying code unpacking tool using dynamorio," https://github.com/BreakingMalware/Selfie.

[10] Karl, "Automated unpacking: A behaviour based approach," https://github.com/malwaremusings/unpacker.

[11] T. Touili and X. Ye, "Reachability analysis of self modifying code," in *ICECCS*. IEEE, 2017, pp. 120–127.

[12] A. Lakhotia, E. U. Kumar, and M. Venable, "A method for detecting obfuscated calls in malicious binaries," *IEEE Trans. Software Eng.*, vol. 31, no. 11, pp. 955–968, 2005. [Online]. Available: https://doi.org/10.1109/TSE.2005.120

[13] F. Song and T. Touili, "Efficient ctl model-checking for pushdown systems," *Theoretical Computer Science*, vol. 549, pp. 127–145, 2014.

[14] T.Touili and X.Ye, "Ctl model checking of self modifying code," https://lipn.univ-paris13.fr/~xin/ctl_full.pdf.

[15] F. Song and T. Touili, "Efficient malware detection using model-checking," in *FM*, 2012, pp. 418–433.

[16] P. Beaucamps, I. Gnaedig, and J.-Y. Marion, "Behavior abstraction in malware analysis," in *International Conference on Runtime Verification*. Springer, 2010, pp. 168–182.

[17] F. Song and T. Touili, "Ltl model-checking for malware detection," in *TACAS*. Springer, 2013, pp. 416–431.

[18] H. Nguyen and T. Touili, "CARET model checking for malware detection," in *SPIN 2017*, H. Erdogmus and K. Havelund, Eds. ACM, 2017, pp. 152–161. [Online]. Available: https://doi.org/10.1145/3092282.3092301

[19] G. Bonfante, J.-Y. Marion, and D. Reynaud-Plantey, "A computability perspective on self-modifying programs," in *SEFM*, 2009, pp. 231–239.

[20] H. Cai, Z. Shao, and A. Vaynberg, "Certified self-modifying code," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 66–77, 2007.

[21] S. K. D. K. P. Coogan and G. M. Townsend, "On the semantics of self-unpacking malware code," Citeseer, Tech. Rep., 2008.

[22] B. Anckaert, M. Madou, and K. De Bosschere, "A model for self-modifying code," in *IH*, 2006, pp. 232–248.

[23] S. Blazy, V. Laporte, and D. Pichardie, "Verified abstract interpretation techniques for disassembling low-level self-modifying code," *Journal of Automated Reasoning*, vol. 56, no. 3, pp. 283–308, 2016.

[24] K. A. Roundy and B. P. Miller, "Hybrid analysis and control of malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2010, pp. 317–338.

[25] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic static unpacking of malware binaries," in *WCRE*, 2009, pp. 167–176.

[26] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *WORM*. ACM, 2007, pp. 46–53.

[27] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *ACSAC*, 2006, pp. 289–300.

[28] I. Walukiewicz, "Model checking ctl properties of pushdown systems," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 2000, pp. 127–138.

[29] V. H. Kinder.J., "Jakstab: A static analysis platform for binaries," in *CAV*. Springer, 2008, pp. 423–427.

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

[31] F. Song and T. Touili, "Pumoc: a ctl model-checker for sequential programs," in *ASE*, 2012, pp. 346–349.

[32] ——, "Efficient malware detection using model-checking," in *International Symposium on Formal Methods*. Springer, 2012, pp. 418–433.

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