Reachability Analysis of Concurrent Self-modifying Code

Walid Messahel, Tayssir Touili

Reachability analysis of concurrent self-modifying code*

Walid Messahel
LIPN, CNRS and University Sorbonne Paris Nord

Tayssir Touili
IRIF, CNRS and University Paris Cité

***Abstract*— We introduce a new effective way to analyze multi-threaded programs that contain self modifying code, i.e., code that modifies itself during its execution. This kind of code is widely used in malware to hide the malicious portions of their code. We introduce a new model called Self Modifying Dynamic Pushdown Network (SM-DPN) to model such programs. A SM-DPN is a network of Self-Modifying Pushdown Systems, i.e., Pushdown Systems that can modify their instructions on the fly during execution. We use finite automata to model regular infinite sets of configurations of SM-DPNs, and propose an algorithm to compute a finite automaton corresponding to the pre* reachable configurations of SM-DPNs. This allows to perform the backward reachability analysis of self modifying multi-threaded programs. We implemented our techniques in a tool and obtained encouraging results. In particular, our tool was able to detect a self-modifying multithreaded malware.**

I. Introduction

Self-modifying code is code that changes its own instructions during execution time, without any external intervention. This technique is used for different purposes: some software engineers use it to protect their products from being reverse engineered (code obfuscation), while malware writers use it to evade detection by anti-malware systems.

On the other hand, concurrent programming allows programs to perform multiple tasks simultaneously to be more efficient. Handling this class of programs is known to be delicate and bug-prone, making the process of automatically analysing such category of programs challenging.

In this work, we tackle the analysis problem of *self-modifying* and *concurrent* programs. Indeed, this kind of programs is nowadays widely used e.g. by malware writers to make their malware more efficient when executed (this is the role of concurrency) and hard to analyse and to detect by antiviruses (this is the role of self-modifying code). This makes the issue more complicated and more challenging, since it invloves two sources of difficulties: *concurrency* and *self-modifying code*.

There are several ways to implement a self modifying code. In this work we consider *direct memory modification*, which consists in modifying the code directly in the execution memory zone, using low-level assembly instructions such as **mov** instructions. Indeed, malware with self modifying code tend to use assembly **mov** instructions that have the ability to access and modify memory enabling the code to alter itself automatically by changing the instruction's binary code stored in memory. Let us consider the example shown in Listing 1.

```
Address    Bytecode      Assembly

0x13         b807        Mov eax,0x7
0x15         89c0        Mov eax,eax
0x17         c602        Mov [0x14],0x2
0x19         c618cd      Mov [0x15],0xcd
0x1c         c61980      Mov [0x16],0x80
0x1f         ff2513      Jmp 0x13
```

Listing 1: Binary code with self modifying instructions and thread creation

```
0x13   b807     Mov eax,0x7   -executing -->    Mov eax,0x2
                                Mov [0x14],0x2

0x15   89c0     Mov eax,eax   -executing -->    int 0x80
                                Mov [0x15],0xcd
                                and
                                Mov [0x16],0x80


0x17   c602     Mov [0x14],0x2
0x19   c618cd   Mov [0x15],0xcd
0x1c   c61980   Mov [0x16],0x80
0x1f   ff2513   Jmp 0x13
```

Listing 2: The code after applying the self-modifying instructions

The first column gives the addresses of the instructions. The binary code is given in the second column, while its corresponding assembly code is given in the right-most column. For example, the binary code **b8 07** stored at address **0x13** corresponds to the instruction **mov eax,0x7**. This piece of code contains *self-modifying code* and *concurrency* (thread creation). The code obtained after executing the self-modifying instructions is shown in Listing 2. Let us explain how this code is both self-modifying and multi-threaded. The instruction **mov [0x14],0x2** will replace the content in the address **0x14** with **0x2** resulting in the new instruction starting from the address **0x13**: **mov eax,0x2**. Thus, **mov [0x14],0x2** is self-modifying. It changes the instruction **mov eax,0x7** at address **0x13** into **mov eax,0x2**. The instruction **mov [0x15],0xcd** will replace the content in the address **0x15** with **0xcd**, and instruction **mov [0x16],0x80** will replace the content in the address **0x16** with **0x80** resulting in the new instruction starting from the address **0x15**: **Int 0x80** (the binary code **cd 80** corresponds to the assembly instruction **int 0x80**). Thus, these instructions **mov [0x15],0xcd** and **mov [0x16],0x80** are both self-modifying. They change the instruction **Mov eax,eax**

*This work was partially funded by the ERGANEO grant MALWARE and the french ANR grant Defmal "ANR-22-PECY-0007"

at address **0x15** into **int 0x80**. This kernel function call **int 0x80** uses the content of the **eax** register as a parameter to identify which function to call. In this case, since **int 0x80** is called while **eax** contains **0x2**, the function **Fork** that creates a new process will be called. Thus, **int 0x80** will create a new thread. Therefore, this piece of code contains *self-modifying code* and *concurrency* (thread creation).

In this work, we consider the reachability analysis of this class of programs (self-modifying programs with thread creation). For this purpose, we need an adequate formalism to model such programs. PushDown Systems (PDS) are known to be a natural model for sequential programs [1], since they allow to mimic the program's stack, and thus to model procedure calls faithfully. Dynamic Pushdown Netwroks (DPNs) [2] were then introduced to model concurrent programs with dynamic thread creation in a precise way. A DPN can be seen as a netwok of pushdown systems, where each process (PDS) can create new processes (PDSs) during its execution. However, DPNs cannot model self-modifying code. On the other hand, Self-Modifying PushDown Systems (SM-PDS) were introduced in [3]–[5] to model sequential programs with self-modifying code. A SM-PDS can be seen as a PDS that can modify its own set of transitions during execution. Following these approaches, in this work we go one step further and propose a new model, called Self-Modifying Dynamic Pushdown Network (SM-DPN), to formally represent concurrent programs that involve self-modifying code as well as dynamic thread creation. Roughly speaking, a SM-DPN is a DPN where each process of the network is a SM-PDS, i.e., a pushdown system with self-modifying instructions.

We show how SM-DPNs can be used to naturally represent self-modifying programs with dynamic thread creation. It turns out that SM-DPNs are equivalent to standard DPNs. We show how to translate a SM-DPN to a standard DPN. This translation is exponential, making the reachability analysis on the equivalent DPN not efficient. Therefore, we propose a *direct* algorithm to compute the backward (Pre^*) reachability sets for SM-DPNs. This allows to efficiently perform reachability analysis for self-modifying concurrent programs. Our algorithm is based on (1) representing regular (potentially infinite) sets of configurations of SM-DPNs using finite state automata, and (2) applying a saturation procedure on the finite state automata in order to take into account the effect of applying the rules of the SM-DPN. We implemented our algorithms in a tool that takes as input either a SM-DPN or a self-modifying binary program with thread creation. Our experiments show that our *direct* techniques are much more efficient than translating the SM-DPN to an equivalent DPN and then applying the standard reachability algorithm for DPNs [2]. To show the efficiency of our approach, we successfuly applied our tool for malware detection.

Outline. The rest of the paper is structured as follows: The related work is decribed in Section 2. Section 3 introduces our new model and shows how to translate a SM-DPN to an equivalent DPN. In Section 4, we give the translation from a self-modifying binary code with thread creation to a SM-DPN. In Section 5, we define finite automata to represent regular (potentially infinite) sets of configurations of SM-DPNs. Section 6 gives our algorithm to compute the backward reachability set of SM-DPNs. Section 7 describes our experiments. The proofs are given in the appendix.

II. Related Work

Analyzing binary code has always been an interesting field of study by a variety of computer scientists especially for security purposes either to disclose vulnerabilities or to detect hidden malware. Model checking and static analysis approaches have been extensively used to analyze binary programs in [6]–[10]. However, these works do not consider self-modifying code.

Analyzing self-modifying code is challenging due to the changing nature of the code. Tha majority of the works that deal with self-modifying code use dynamic analysis like [11]–[13]. However, dynamic analysis cannot cover all possible execution traces of the program. There are several works that perform static analysis of self-modifying code. [14] use separation logic to describe self-modifying code. However, [14] requires programs to be manually annotated with invariants. [15] propose a formal semantics for self-modifying codes. This work deals only with packing and unpacking behaviours. Bonfante et al. [16] 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 [14]–[16] are too abstract to be used in practice. [17] 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 that keeps track of the possible states that programs can reach, and with edges that can be conditional on the state of the target memory location. This SE-CFG representation does not allow to take into account the stack of the program. [18] 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. [19] combine static and dynamic analysis techniques to analyse self-modifying programs. These techniques [18], [19] cannot handle the program's stack.

Self Modifying PushDown Systems (SM-PDSs) were introduced in [3] and were successufully applied for the analysis of *sequential* self-modifying code [3]–[5]. Our work can be seen as an extension of this approach to *concurrent* self-modifying code.

On the other hand, Dynamic Pushdown Networks (DPNs) [2], [20], [21] and its extensions [22]–[26] were broadly used to analyse concurrent programs with thread creation. All these works cannot deal with self-modifying code. Other models based on networks of pushdown systems [27]–[31] were extensively used to deal with concurrent programs. However, these works cannot handle thread creation, nor self-modifying code.

III. Self Modifying Dynamic Pushdown Network

A. Definition

We introduce in this section our new model: Self Modifying Dynamic Pushdown Network. A Self Modifying Pushdown System (SM-PDS) [3] is a pushdown system that has the ability to modify its own set of rules during the execution. A Self Modifying Dynamic Pushdown Network (SM-DPN) consists of a network of SM-PDSs that is capable of modeling a collection of pushdown processes running in parallel, where each of these pushdown processes can dynamically change its current set of rules and create new processes during its execution. Intuitively, a SM-DPN is a DPN [2] (a network of pushdown processes) where each process of the network has the ability to modify its own set of transitions. Formally:

Definition 1. *A Self Modifying Dynamic Pushdown Network (SM-DPN) is a tuple $\Re = (P, \Gamma, \Delta, \Delta_c)$, where P is a finite set of control points, Γ is a finite set of stack symbols, $\Delta \subseteq (P\times\Gamma)\times(P\times\Gamma^*)\cup(P\times\Gamma)\times(P\times\Gamma^*)\times((P\times 2^{\Delta\cup\Delta_c})\times\Gamma^*)$ is a finite set of transitions, and $\Delta_c \subseteq P\times(\Delta\cup\Delta_c)\times(\Delta\cup\Delta_c)\times P$ is a finite set of modifying transitions rules. A Dynamic Pushdown Network (DPN) is a SM-DPN with $\Delta_c = \emptyset$.*

Every process in the network has its current set of transition rules θ called the phase, such that $\theta \subseteq \Delta \cup \Delta_c$. Rules in Δ_c can alter a process phase. We can distinguish three different types of transition rules used by the SM-DPN:

- $((p,\gamma),(p_0,w_0)) \in \Delta$ where $p, p_0 \in P$, $\gamma \in \Gamma$, $w_0 \in \Gamma^*$. This rule can also be written as $p\gamma \hookrightarrow p_0 w_0 \in \Delta$.
  This rule means that if a process of the network is in control point p with γ as its top element of the stack then it can move to control point p_0, pop γ and push w_0.
- $((p,\gamma),(p_1,w_1),((p_0,\theta),w_0)) \in \Delta$ where $p, p_0, p_1 \in P$, $\gamma \in \Gamma, w_0, w_1 \in \Gamma^*$, $\theta \subseteq \Delta \cup \Delta_c$. This rule can also be written as $p\gamma \hookrightarrow p_1 w_1 \triangleright (p_0,\theta) w_0 \in \Delta$.
  This rule means that if a process of the network is in control point p with γ as its topmost stack element, then it can move to control point p_1, pop γ, push w_1 and create a new process in the network having p_0 as its initial control point, w_0 as its initial stack content and θ as its initial current set of rules (phase).
- $(p, r_1, r_2, p_0) \in \Delta_c$ where $p, p_0 \in P$, $r_1, r_2 \in \Delta \cup \Delta_c$. This rule can also be written as $p \xrightarrow{(r_1,r_2)} p_0 \in \Delta_c$.
  This rule means that if a process of the network is in control point p and r_1 is in its current set of rules, then it can move to control point p_0 and update its current set of transition rules by replacing the rule r_1 with the rule r_2.

A local configuration of a process of the network can be represented by $(p,\theta)w$, where $p \in P$ is the control point of the process, $\theta \subseteq \Delta \cup \Delta_c$ is its current set of rules (phase), $w \in \Gamma^*$ is its stack content. For simplicity, from now on, we will sometimes write p^θ instead of (p,θ).

A global SM-DPN configuration is a word of the form $p_0^{\theta_0} w_0 p_1^{\theta_1} w_1 \dots p_n^{\theta_n} w_n$ where $p_0, p_1, \dots, p_n \in P$, $w_0, w_1, \dots, w_n \in \Gamma^*$ and $\theta_0, \theta_1, \dots, \theta_n \subseteq \Delta \cup \Delta_c$. This word means that there are n running processes in the network and for every i such as $0 \le i \le n$, the process i is in control point p_i, with w_i as its stack content and have θ_i as its current set of rules.

Let $Conf_\Re$ be the set of all global configurations of the SM-DPN $\Re$. We define the transition relation $\Rightarrow_\Re$ to be the smallest relation between two configurations in $Conf_\Re \times Conf_\Re$ as follows :

- Let $c = up^\theta wv$, $c' = up'^{\theta'} wv$ with $u, v \in Conf_\Re$, $w \in \Gamma^*$, if $r = p \xrightarrow{(r_1,r_2)} p' \in \Delta_c \cap \theta$, $r_1 \in \theta$ and $\theta' = (\theta \backslash \{r_1\}) \cup \{r_2\}$ then c is a predecessor of c' (also written as $c \Rightarrow_\Re c'$). The rule r moves the process from the control point p to the control point p' and changes the current phase (current set of transition rules) by removing r_1 and replacing it with r_2 without altering the content of the stack.
- Let $c = up^\theta \gamma wv, c' = up'^\theta w' wv$ with $\gamma \in \Gamma$, $w, w' \in \Gamma^*$, $u, v \in Conf_\Re$, and $r = p\gamma \hookrightarrow p'w' \in \Delta \cap \theta$, then $c \Rightarrow_\Re c'$. The rule r moves the SM-DPN process from the control point p to the control point p', pops γ from the stack and pushes w' into the stack. This rule maintains the current phase (θ) untouched.
- Let $c = up^\theta \gamma wv$, $c' = up_1^{\theta_1} w_1 p'^\theta w' wv$ with $\gamma \in \Gamma$, $w \in \Gamma^*$, $u, v \in Conf_\Re$, if $r = p\gamma \hookrightarrow p'w' \triangleright p_1^{\theta_1} w_1 \in \Delta \cap \theta$, then $c \Rightarrow_\Re c'$. Here the rule r will move the SM-DPN process from the control point p to the control point p', pop γ from the stack, push w' into the stack and create a new process on the control point p_1, with w_1 as stack content, θ_1 as the initial phase and maintains the current phase (θ) untouched.

We define $\Rightarrow_\Re^*$ as the transitive reflexive closure of $\Rightarrow_\Re$. If a configuration c' is reachable from c_0 in i steps by applying $\Rightarrow_\Re$ i times, we write $c_0 \Rightarrow_\Re^i c'$.

We denote the set of immediate predecessors (resp. successors) of a configuration c as $Pre_\Re(c) = \{c_1 \in Conf_\Re : c_1 \Rightarrow_\Re c\}$ (resp. $Post_\Re(c) = \{c_1 \in Conf_\Re : c \Rightarrow_\Re c_1\}$). Let $Pre_\Re^*$(resp. $Post_\Re^*$) denote the reflexive transitive closure of $Pre_\Re$ (resp. $Post_\Re$). These notations can be generalized to sets of configurations in the obvious ways.

B. From SM-DPN to DPN

Since the number of phases is finite, a SM-DPN can be simulated by a DPN [2] by encoding the phases along with the DPN control points.

Let $\Re = (P, \Gamma, \Delta, \Delta_c)$ be a SM-DPN, we compute the corresponding DPN $M = (P', \Gamma, \Delta')$ where P' is the DPN set of control points, Δ' is the DPN set of transition rules such that $P' = P \times 2^{\Delta\cup\Delta_c}$, and Δ' is computed as follows:

Initially $\Delta' = \emptyset$. For every $\theta \in 2^{\Delta\cup\Delta_c}$ and $r \in \theta$.

1) if $r = p\gamma \hookrightarrow p'w' \in \Delta \cap \theta$, then $(p,\theta)\gamma \hookrightarrow (p',\theta)w' \in \Delta'$.
2) if $r = p\gamma \hookrightarrow p'w' \triangleright (p_2,\theta_2)w_2 \in \Delta \cap \theta$, then $(p,\theta)\gamma \hookrightarrow (p',\theta)w' \triangleright (p_2,\theta_2)w_2 \in \Delta'$.

3) if $r = p \xhookrightarrow{(r_1,r_2)} p' \in \Delta_c \cap \theta$, then for every $\gamma \in \Gamma$, $(p,\theta)\gamma \hookrightarrow (p',\theta')\gamma \in \Delta'$, where $\theta' = (\theta \setminus \{r_1\}) \cup \{r_2\}$.

We can show that:

Proposition 1. *Let $u, v \in Conf_{\Re}$, $u \Rightarrow_{\Re} v$ iff $u \Rightarrow_M v$.*

Thus, we get:

Theorem 1. *Let $\Re = (P, \Gamma, \Delta, \Delta_c)$ be a SM-DPN, we can compute a corresponding DPN $M = (P', \Gamma, \Delta')$ such that $|P'| = |P| \cdot 2^{\mathcal{O}(|\Delta|+|\Delta_c|)}$ and $|\Delta'| = (|\Delta| + |\Delta_c| \cdot |\Gamma|) \cdot 2^{\mathcal{O}(|\Delta|+|\Delta_c|)}$.*

IV. Modeling self modifying concurrent code with SM-DPN

We show in this section how to build a SM-DPN from a binary program with self-modifying code and dynamic thread creation. We suppose we are given an oracle $\mathcal{O}$ that extracts from the binary code a corresponding assembly program, together with informations about the values of the registers and the memory locations at each control point of the program. We could use Jakstab [32], IDA Pro [33] or radare2 [34] to get this oracle. We translate the assembly program into a SM-DPN whose control points are the control points of the binary program and such that the stacks of the different pushdown processes mimic the program's processes stacks. The non self-modifying instructions of the program define the rules Δ of the SM-DPN (which are standard DPN rules), and can be obtained following the translation of [10] that models non self-modifying sequential instructions of the program by a PDS. Thread creation instructions can then be represented by rules of the form $p\gamma \hookrightarrow p_1 w_1 \triangleright (p_0, \theta) w_0 \in \Delta$, where $(p_0, \theta) w_0$ corresponds to the initial configuration of the newly created thread.

As for the self-modifying instructions of the program, we use the translation of [3]. These instructions define the set of changing rules Δ_c. As mentionned in the introduction, in this work we tackle self-modifying instructions of the form $mov\ l\ v$, where l is a location of the program that stores executable data. This instruction replaces the value at location l (in the binary code) with the value v. Let i_1 be the initial instruction involving the location l, and let i_2 be the new instruction involving the location l, after applying the $mov\ l\ v$ instruction. As in [3], we assume that i_1 and i_2 have the same number of operands (to ensure that only one instruction is modified). Let r_1 (resp. r_2) be the SM-DPN rule corresponding to the instruction i_1 (resp. i_2). Suppose from control point n to n', we have this $mov\ l\ v$ instruction, then we add $n \xhookrightarrow{(r_1,r_2)} n'$ to Δ_c. This is the SM-DPN rule corresponding to the instruction $mov\ l\ v$ at control point n.

V. Representing infinite sets of configurations of SM-DPN

Following [2], to finitely represent a regular infinite set of SM-DPN configurations, we use a special kind of automata called $\Re$-automata.

Let $\Re = (P, \Gamma, \Delta, \Delta_c)$ be a SM-DPN. a $\Re$-automaton is a tuple $A = (S, \Omega, T, s^0, F)$ with the following conditions :

1) $\Omega = (P \times 2^{\Delta \cup \Delta_c}) \cup \Gamma$ is the automaton alphabet.
2) S is a finite set containing the automaton states partitioned into two subsets S_c and S_s s.t $S = S_c \cup S_s, S_c \cap S_s = \emptyset$ and for every $s \in S_c$ and every p^θ such that $p \in P$, $\theta \subseteq \Delta \cup \Delta_c$, there is a unique state called $s_{p^\theta} \in S_s$.
3) There is a relation $T' \subseteq S_s \times \Gamma \times (S_s \setminus \{s_{p^\theta} : s \in S_c, p \in P, \theta \subseteq \Delta \cup \Delta_c\}) \cup S_s \times \{\epsilon\} \times S_c$ such that $T = T' \cup \{(s, p^\theta, s_{p^\theta}) : s \in S,\ p \in P,\ \theta \subseteq \Delta \cup \Delta_c\}$.
4) $s^0 \in S_c$ is the initial state.
5) $F \subseteq S$ is the set of final states .

Note that condition (3) implies the following properties:

- For each $p \in P$, $\theta \subseteq \Delta \cup \Delta_c$, $s \in S_c$, s is the only predecessor of s_{p^θ}.
- States s in S_c do not have Γ-transitions.
- Only ϵ-moves from states in S_s lead to states in S_c.
- States s in S_s do not have p-successors, for $p \in P$

For $y \in \Gamma \cup \{\epsilon\}$ and $s, s' \in S$, if $(s, y, s') \in T$ we write $s \xrightarrow{y}_T s'$. This notation can be extended in the obvious manner to sequences of symbols as follows :

$\forall s \in S,\ s \xrightarrow{\epsilon}_T s$ and $\forall s, s' \in S,\ \forall y \in \Gamma \cup \{\epsilon\},\ \forall w \in \Gamma^*,\ s \xrightarrow{yw}_T s'$ iff $\exists s" \in S$ such that $s \xrightarrow{y}_T s" \xrightarrow{w}_T s'$. We will remove the subscript T if it is understood in the context.

Intuitively, the conditions above make sure that every path in the $\Re$-automaton is the concatenation of paths of the form $s^0 \xrightarrow{p_0^{\theta_0}} s^0_{p_0^{\theta_0}} \xrightarrow{w_0} q_0 \xrightarrow{\epsilon} s^1 \xrightarrow{p_1^{\theta_1}} s^1_{p_1^{\theta_1}} \xrightarrow{w_1} q_1 \xrightarrow{\epsilon} \dots \xrightarrow{\epsilon} s^n \xrightarrow{p_n^{\theta_n}} s^n_{p_n^{\theta_n}} \xrightarrow{w_n} q_n$ such that $s^0, s^1 \dots s^n \in S_c$, $s^0_{p_0^{\theta_0}}, s^1_{p_1^{\theta_1}} \dots s^n_{p_n^{\theta_n}} \in S_s$, $q_0, q_1 \dots q_n \in S_s$, $p_0, p_1 \dots p_n \in P$, $\theta_0, \theta_1 \dots \theta_n \subseteq \Delta \cup \Delta_c$, $w_0, w_1 \dots w_n \in \Gamma^*$.

A configuration $p_0^{\theta_0} w_0 p_1^{\theta_1} w_1 \dots p_n^{\theta_n} w_n$ is accepted by an automaton A if there exists a path of the form $s^0 \xrightarrow{p_0^{\theta_0}} s^0_{p_0^{\theta_0}} \xrightarrow{w_0} q_0 \xrightarrow{\epsilon} s^1 \xrightarrow{p_1^{\theta_1}} s^1_{p_1^{\theta_1}} \xrightarrow{w_1} q_1 \dots s^n \xrightarrow{p_n^{\theta_n}} s^n_{p_n^{\theta_n}} \xrightarrow{w_n} q_f$ such that $q_f \in F$. We denote by $L(A)$ the set of configurations accepted by A. A set of global SM-DPN configurations $\mathcal{L}$ is regular if there exists an $\Re$-automaton A such that $\mathcal{L} = L(A)$.

A. An example

Consider the regular set of configurations $C = (p_0^{\theta_0} \gamma_1^+ \gamma_2 p_1^{\theta_1} \gamma_1)^*$ of a SM-DPN. This set can be represented by the automaton $A = (S, \Omega, T, s^0, F)$ presented in Figure 1. As can be seen, the structure and the different constraints imposed to the automaton ensure that the configurations of different processes of the network are linked by ϵ transitions in the automaton.

VI. SM-DPN reachability analysis

It has been shown in [2] that $Post^*$ images of regular sets of configurations are not regular for DPNs. Since SM-DPNs

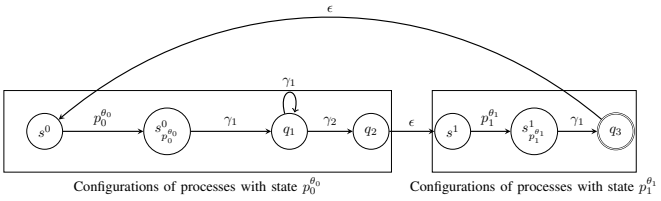

Fig. 1: Example of a SM-DPN automaton

are equivalent to DPNs, this holds for SM-DPNs as well. Thus, we concentrate on Pre^* images in this section. Let then $\Re = (P, \Gamma, \Delta, \Delta_c)$ be a SM-DPN. Since this SM-DPN can be converted to a DPN, we can use the Pre^* algorithm for DPNs, presented in [2], to perform the reachability analysis of the SM-DPN. However, this approach is very expensive in both memory consumption and time of execution as we will show in Section VII. Thus we present here a *direct* algorithm to perform Pre^* images for SM-DPN.

Let $A = (S, \Omega, T, s^0, F)$ be an $\Re$-automaton. We build $A_{Pre^*} = (S, \Omega, T', s^0, F)$ by adding new transitions to the automaton A using the following saturation rules. Initially $T' = T$:

- β_1 : if $r = p_0\gamma \hookrightarrow p_1 w_1 \in \Delta$, where $w_1 \in \Gamma^*, \gamma \in \Gamma$, then for every $\theta \subseteq \Delta \cup \Delta_c$ such that $r \in \theta$, if there exists a path in the form $s \xrightarrow{p_1^\theta w_1}_{T'} s'$, for $s \in S_c$, $s' \in S$, then $(s_{p_0^\theta}, \gamma, s') \in T'$.

- β_2 : if $r = p_0\gamma \hookrightarrow p_1 w_1 \triangleright p_2^{\theta_2} w_2 \in \Delta$ where $w_1, w_2 \in \Gamma^*, \gamma \in \Gamma$, then for every $\theta \subseteq \Delta \cup \Delta_c$ such that $r \in \theta$, if there exists a path in the form $s \xrightarrow{p_2^{\theta_2} w_2 p_1^\theta w_1}_{T'} s'$, for $s \in S_c$, $s' \in S$, then $(s_{p_0^\theta}, \gamma, s') \in T'$.

- β_3 : if $r = p_0 \xhookrightarrow{(r_1, r_2)} p_1 \in \Delta_c$, then for every $\theta \subseteq \Delta \cup \Delta_c$ such that $r_2 \in \theta$, for every $\gamma \in \Gamma$, if there exists a path $s \xrightarrow{p_1^\theta \gamma}_{T'} s'$ for $s \in S_c$, $s' \in S$, then $(s_{p_0^{\theta_0}}, \gamma, s') \in T'$ with $\theta_0 = (\theta \setminus \{r_2\}) \cup \{r_1\}$.

The procedure above terminates since there is a finite number of states and phases. Let us explain the intuition behind these rules:

- Let $c = up_0^\theta \gamma v$ and $c' = up_1^\theta w_1 v$ be two global configurations and $r = p_0\gamma \hookrightarrow p_1 w_1 \in \Delta$ be a SM-DPN transition rule. If $r \in \theta$, then c is a predecessor of c'. The intuition behind β_1 is that if c' is accepted by the automaton A_{Pre^*}, then c should also be accepted by A_{Pre^*}, i.e if there exists a path that accepts c' such that $s^0 \xrightarrow{u}_{T'} s \xrightarrow{p_1^\theta}_{T'} s_{p_1^\theta} \xrightarrow{w_1}_{T'} s' \xrightarrow{v}_{T'} q_f, q_f \in F$, then, β_1 adds the new transition $(s_{p_0^\theta}, \gamma, s')$ to T'. Thus we will get a new path in the automaton such that $s^0 \xrightarrow{u}_{T'} s \xrightarrow{p_0^\theta}_{T'} s_{p_0^\theta} \xrightarrow{\gamma}_{T'} s' \xrightarrow{v} q_f, q_f \in F$. This path accepts the configuration c then, c is accepted by the automaton A_{Pre^*}.

- Let $r = p_0\gamma_0 \hookrightarrow p_1 w_1 \triangleright p_2^{\theta_0} w_2 \in \Delta$, consider now the configurations $c = up_0^\theta \gamma_0 v$ and $c' = up_2^{\theta_0} w_2 p_1^\theta w_1 v$, c is a predecessor of c'. The intuition behind β_2 is that if c' is accepted by the automaton A_{Pre^*}, then c should also be accepted by A_{Pre^*}, i.e if there exists a path that accepts c' such that $s^0 \xrightarrow{u} s^1 \xrightarrow{p_2^{\theta_0}} s^1_{p_2^{\theta_0}} \xrightarrow{w_2} q_1 \xrightarrow{\epsilon} s^2 \xrightarrow{p_1^\theta} s^2_{p_1^\theta} \xrightarrow{w_1} q' \xrightarrow{v} q_f, q_f \in F$. By the saturation rule β_2, we add the new transition $(s^1_{p_0^\theta}, \gamma_0, q')$ to T'. Thus we will get a new path in the automaton in such a way $s^0 \xrightarrow{u} s^1 \xrightarrow{p_0^\theta} s^1_{p_0^\theta} \xrightarrow{\gamma_0} q' \xrightarrow{v} q_f, q_f \in F$. This path accepts the configuration c then, c is accepted by the automaton A_{Pre^*}.

- Let $r = p_0 \xhookrightarrow{(r_1, r_2)} p_1 \in \Delta_c$ and $c = up_0^\theta \gamma_0 v$, $c' = up_1^{\theta_0} \gamma_0 v$. The intuition behind the saturation rule β_3 is that if $r, r_2 \in \theta_0$ and $\theta = (\theta_0 \setminus \{r_2\}) \cup \{r_1\}$ then c is a predecessor of c'. Thus, if c' is accepted by the automaton A_{Pre^*} then so should be c. Thus, if there there exists a path in the automaton of the form $s^0 \xrightarrow{u} s \xrightarrow{p_1^{\theta_0}} s_{p_1^{\theta_0}} \xrightarrow{\gamma_0} q' \xrightarrow{v} q_f, q_f \in F$. By applying β_3 we add a new transition $(s_{p_0^\theta}, \gamma_0, q')$ to T'. Thus we will get a new path in the automaton in such a way $s^0 \xrightarrow{u} s \xrightarrow{p_0^\theta} s_{p_0^\theta} \xrightarrow{\gamma_0} q' \xrightarrow{v} q_f, q_f \in F$ which means c is accepted by the automaton A_{Pre^*}.

Thus, we can show that :

Theorem 2.

$$L(A_{Pre^*}) = Pre^*(L(A)).$$

To prove Theorem 2 we show the following two directions :

Lemma 1. $Pre^*(L(A)) \subseteq L(A_{Pre^*})$

Lemma 2. $L(A_{Pre^*}) \subseteq Pre^*(L(A))$

VII. Experiments

We implemented our algorithms in a tool. Our experiments were conducted on a machine with 11th Gen Intel® Core™ i7-11850H @ 2.50GHz × 16 CPU and 30,6 GiB of Memory. Our experiments contain two parts: (1) First, we show the efficiency of our approach vs. the approach that consists in translating the SM-DPN to a DPN and then applying the standard Pre^* algorithm of [2]. (2) Then, we show how our approach can be used for malware detection.

A. Our algorithm vs. the standard Pre^* algorithm of DPNs

To practically prove the efficiency of our Pre^* computation compared to the approach that consists in translating the SM-DPN to an equivalent DPN and then apply the standard Pre^* algorithm of [2], we have conducted a number of experiments. To realize such experimental study we developed a script that randomly generates a number of SM-DPNs then converts them

to equivalent DPNs as described in Section 3. The results of the comparision using the Pre^* algorithm are reported in Table 1. **Column** $|\Delta|+|\Delta_c|$ gives the total number of rules of the SM-DPN. **Column** *Our SM-DPN Algo* gives the execution time and the memory consumption of our Pre^* SM-DPN algorithm. **Column** SM-DPN $\Rrightarrow$ DPN gives the conversion time it takes to translate the SM-DPN to a corresponding DPN. **Column** *DPN Results* gives the execution time of the standard Pre^* DPN algorithm of [2]. **Column** *DPN Total* gives the total time it takes to convert the SM-DPN to an equivalent DPN and then apply the Pre^* DPN algorithm of [2] (conversion + Pre^* execution time). As can be seen, our *direct* algorithm has shown a remarkable improvement in the time of execution and the memory consumption. For example, with the configuration $|\Delta|+|\Delta_c| = 3568+14$ we can notice the huge difference between the time taken by our SM-DPN *direct* Pre^* algorithm which is 4446.49s and the time taken by the DPN standard Pre^* algorithm of [2] (15540s: more than 4.3 Hours). Also we notice a significant memory consumption improvement like the results of the example with the configuration $|\Delta| + |\Delta_c| = 522 + 14$ where our SM-DPN algorithm used 18.05MB, while the DPN algorithm has required 1280.08MB of memory.

B. Application to malware detection

Malware writers extensively use (1) self-modifying code as an obfuscation technique and (2) thread creation to make their malware more efficient and thus harder to be discovered. Thus, we apply our tool for malware detection. We consider a concurrent self modifying binary file infected by the BadRabbit ransomware which is a notorious malware that performs a drive-by attack to install itself through fake adobe flash installer or updates, encrypts all data and uses EternalRomance exploit to spread within the corporate network [35]. The binary "BadRabbit.exe" is obtained from **MalwareCollection** github repo [36]. The malicious behavior is unreachable if one does not take into account the *self-modifying* nature of the *concurrent* code : after executing the self-modifying code, the control point will jump to the part containing the malicious behavior. We use radare2 [34] to decompile the binary and retrieve its assembly code. Then, we model the corresponding assembly program by a SM-DPN as explained in Section IV. Finally, we apply our Pre^* algorithm to show that the malicious part of the code is reachable from its entry point. Thus, applying our tool, we deduce that the code is malicious.

REFERENCES

[1] J. Esparza and S. Schwoon, "A bdd-based model checker for recursive programs," in *CAV*, 2001.

[2] A. Bouajjani, M. Müller-Olm, and T. Touili, "Regular symbolic analysis of dynamic networks of pushdown systems," in *International Conference on Concurrency Theory*. Springer, 2005, pp. 473–487.

[3] T. Touili and X. Ye, "Reachability analysis of self modifying code," in *22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017*. IEEE Computer Society, 2017, pp. 120–127.

[4] ——, "LTL model checking of self modifying code," in *24th International Conference on Engineering of Complex Computer Systems, ICECCS 2019, Guangzhou, China, November 10-13, 2019*, J. Pang and J. Sun, Eds. IEEE, 2019, pp. 1–10.

[5] ——, "CTL model checking of self modifying code," in *25th International Conference on Engineering of Complex Computer Systems, ICECCS 2020, Singapore, October 28-31, 2020*, Y. Li and A. W. Liew, Eds. IEEE, 2020, pp. 11–20.

[6] 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.

[7] G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. Yong, C.-H. Chen, and T. Teitelbaum, "Model checking x86 executables with codesurfer/x86 and wpds++," in *International Conference on Computer Aided Verification*. Springer, 2005, pp. 158–163.

[8] P. K. Singh and A. Lakhotia, "Static verification of worm and virus behavior in binary executables using model checking," in *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, 2003.

[9] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2005.

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

[11] S. Dawei, L. Delong, and Y. Zhibin, "Dynamic self-modifying code detection based on backward analysis," in *Proceedings of the 2018 10th International Conference on Computer and Automation Engineering*, ser. ICCAE 2018, 2018.

[12] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 659–673.

[13] W. Guizani, J.-Y. Marion, and D. Reynaud-Plantey, "Server-side dynamic code analysis," in *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, 2009, pp. 55–62.

[14] H. Cai, Z. Shao, and A. Vaynberg, "Certified self-modifying code," *ACM SIGPLAN Notices*, 2007.

[15] S. K. Debray, K. P. Coogan, and G. M. Townsend, "On the semantics of self-unpacking malware code," in *Citeseer*, 2008.

[16] G. Bonfante, J.-Y. Marion, and D. Reynaud-Plantey, "A computability perspective on self-modifying programs," in *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*, 2009.

[17] B. Anckaert, M. Madou, and K. De Bosschere, "A model for self-modifying code," in *International Workshop on Information Hiding*, 2006.

[18] S. Blazy, V. Laporte, and D. Pichardie, "Verified abstract interpretation techniques for disassembling low-level self-modifying code," *Journal of Automated Reasoning*, 2016.

[19] K. A. Roundy and B. P. Miller, "Hybrid analysis and control of malware," in *International Workshop on Recent Advances in Intrusion Detection*, 2010.

[20] H. Nguyen and T. Touili, "CARET analysis of multithreaded programs," in *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR*, F. Fioravanti and J. P. Gallagher, Eds., 2017.

[21] F. Song and T. Touili, "Model checking dynamic pushdown networks," *Formal Aspects Comput.*, vol. 27, no. 2, pp. 397–421, 2015.

[22] A. Wenner, "Weighted dynamic pushdown networks," in *Programming Languages and Systems, 19th European Symposium on Programming, ESOP*, A. D. Gordon, Ed., 2010.

[23] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek, "Reachability analysis of multithreaded software with asynchronous communication," in *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference*, 2005.

[24] M. Diaz and T. Touili, "Model checking dynamic pushdown networks with locks and priorities," in *Networked Systems - 6th International Conference, NETYS*, 2018.

[25] ——, "Dealing with priorities and locks for concurrent programs," in *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA*, D. D'Souza and K. N. Kumar, Eds., 2017.

[26] ——, "Reachability analysis of dynamic pushdown networks with priorities," in *Networked Systems - 5th International Conference, NETYS*, A. E. Abbadi and B. Garbinato, Eds., 2017.

$\|\Delta\| + \|\Delta_c\|$	**Our SM-DPN Algo**	**SM-DPN $\Rightarrow$ DPN**	**DPN Results**	**DPN Total**
400 + 14	117.63s & 18.08MB	0.19s & 36.09MB	231.61s	231.80s
600 + 12	73.51s & 18.08MB	1.35s & 72.08MB	318.41s	319.76s
2500 + 9	76.57s & 0.45MB	1.80s & 0.45MB	12047.41s	12049.22s
44 + 7	0.09s & 0.45MB	0.046s & 0.45MB	0.56s	0.60s
220 + 8	243.21s & 0.45MB	0.24s & 0.45MB	702.74s	702.99s
10 + 9	0.003s & 0.45MB	0.0722s & 0.45MB	0.1574s	0.22s
850 + 12	729.13s & 36.09MB	9.59s & 288.08MB	1264.47s	1274.07s
50 + 14	1.8492s & 2.21MB	0.2324s & 36.09MB	3108.86s	3109.1007s
140 + 8	224.02s & 0.45MB	0.08s & 0.45MB	1847.18s	1847.2763s
90 + 10	12.28s & 4.58MB	0.33s & 72.08MB	81.19s	81.5271s
1200 + 12	13.21s & 36.09MB	0.58s & 36.09MB	5787.19s	5787.77s
520 + 9	8.44s & 18.08MB	0.07s & 18.08MB	95.89s	95.967s
1000 + 10	48.41s & 36.09MB	5.974s & 72.08MB	146.16s	152.13s
900 + 20	75.5486s & 36.09MB	4794.896s & 2560.09MB	3859.33s	8654.23s
1600 + 13	31.17s & 72.08MB	799.46s & 2560.09MB	1392.90s	2192.36s
1400 + 13	19.42s & 72.08MB	116.17s & 576.09MB	223.31s	339.48s
1100 + 13	32.89s & 36.09MB	95.05s & 576.09MB	126.5723s	221.6304s
800 + 12	334.89s & 36.09MB	14615.336s & 10240.09MB	7223.04s	21838.38s
650 + 12	40.98s & 18.08MB	2734.209s & 2560.09MB	2255.87s	4990.08s
350 + 18	15.49s & 18.08MB	2576.876s & 2560.09MB	17427.96s	20004.83s
1300 + 15	133.52s & 36.09MB	4592.42s & 5120.08MB	2317.41s	6909.83s
700 + 20	408.94s & 36.09MB	259.71s & 1280.08MB	17334.025s	17593.73s
950 + 12	20.27s & 36.09MB	54.37s & 576.09MB	298.16s	352.53s
450 + 13	152.64s & 18.08MB	8410.54s & 5120.08MB	9719.08s	18129.63s
300 + 15	11.01s & 9.09MB	592.95s & 2560.09MB	4174.48s	4767.4428s
2000 + 16	6.43s & 72.08MB	177.087s & 1280.08MB	140.43s	317.52s
4244 + 8	887.33s & 144.09MB	47.45s & 576.09MB	1332.1188s	1379.5702s
2348 + 14	1219.36s & 72.08MB	341.98s & 1280.08MB	3921.81s	4263.80s
2846 + 10	973.38s & 144.09MB	105.33s & 576.09MB	1717.64s	1822.98s
2466 + 6	1712.61s & 72.08MB	203.66s & 1280.08MB	5776.01s	5979.67s
2014 + 14	547.006s & 72.08MB	43.50s & 576.09MB	1082.05s	1125.559s
3986 + 13	11054.64s & 144.09MB	3743.176s & 5120.08MB	85162.2779s	88905.45s
714 + 12	696.03s & 36.09MB	18.26s & 288.08MB	1814.77s	1833.03s
982 + 12	5206.48s & 36.09MB	1008.14s & 1280.08MB	28784.3065s	29792.4513s
2496 + 11	2481.03s & 72.08MB	504.82s & 2560.09MB	8365.26s	8870.09s
2974 + 14	848.19s & 144.09MB	91.57s & 576.09MB	1863.164s	1954.73s
2148 + 13	605.627s & 72.08MB	63.36s & 576.09MB	1700.17s	1763.53s
1998 + 7	1063.92s & 72.08MB	88.14s & 576.09MB	3216.34s	3304.48s
522 + 14	6321.78s & 18.05MB	816.55s & 1280.08MB	21265.877s	22082.42s
3100 + 13	7044.20s & 144.09MB	2575.38s & 5120.08MBs	74293.04s	76868.43
1788 + 12	699.28s & 72.08MB	350.74s & 1280.08MB	3164.87s	3515.61s
874 + 16	5618.36s & 36.09MB	2036.18s & 2560.09MB	46751.55s	48787.74s
2746 + 15	7043.74s & 144.09MB	2010.69s & 2560.09MB	34790.60s	36801.30s
966 + 13	1092.55s & 36.09MB	196.13s & 576.09MB	3862.98s	4059.12s
2916 + 14	1202.16s & 144.09MB	469.23s & 1280.08MB	5241.58s	5710.827s
3568 + 14	4446.49s & 144.09MB	287.94s & 1280.08MB	15252.92s	15540.86s
954 + 14	70629.60s & 36.09MB	17.58s & 144.09MB	5927.26s	5944.84s
2500 + 11	4993.79s & 72.08MB	506.52s & 2560.09MB	87936.67s	88443.19s
2632 + 13	2809.73s & 72.08MB	153.15s & 1280.08MB	12487.38s	12640.54s

TABLE I: SM-DPN vs DPN comparison table.

[27] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," in *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS*, N. Halbwachs and L. D. Zuck, Eds., 2005.

[28] A. Pommellet and T. Touili, "LTL model-checking for communicating concurrent programs," in *Verification and Evaluation of Computer and Communication Systems - 12th International Conference, VECoS*.

[29] ——, "Static analysis of multithreaded recursive programs communicating via rendez-vous," in *Programming Languages and Systems - 15th Asian Symposium, APLAS*, 2017.

[30] N. Kidd, P. Lammich, T. Touili, and T. W. Reps, "A decision procedure for detecting atomicity violations for communicating processes with locks," *Int. J. Softw. Tools Technol. Transf.*, vol. 13, no. 1, pp. 37–60, 2011.

[31] A. Bouajjani, J. Esparza, and T. Touili, "A generic approach to the static analysis of concurrent programs with procedures," in *Conference Record*

of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, A. Aiken and G. Morrisett, Eds., 2003.
[32] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *International Conference on Computer Aided Verification*, 2008.
[33] "IDA Pro," http://www.hex-rays.com/idapro/.
[34] A. K. R. Adrian Studer, Ahmed Mohamed Abd El-MAwgood, "The official radare2 book," https://book.rada.re/credits/credits.html, 2023.
[35] A. Perekalin, "Bad rabbit: A new ransomware epidemic is on the rise," https://www.kaspersky.com/blog/bad-rabbit-ransomware/19887/, 2017.
[36] E. xcp3r, "Malwarecollection," https://github.com/xcp3r/MalwareCollection, 2022.