Analyzing a Concurrent Self-Modifying Program: Application to Malware Detection*

Walid Messahel[1] and Tayssir Touili[2]
[1]*LIPN, CNRS and University Sorbonne Paris Nord, France*
[2]*IRIF, CNRS and University Paris Cité, France*

Keywords: Model Checking, Self-Modifying Code, Pushdown Systems, Malware Detection.

Abstract: We tackle the analysis problem of multi-threaded parallel programs that contain self modifying code, i.e., code that have the ability to reconstruct itself during the execution time. This kind of code is usually used to hide malicious portions of codes so that they cannot be detected by anti-viruses. In (Messahel and Touili, 2024), we introduced 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 proposed an algorithm to perform the backward reachability analysis of SM-DPNs. However, in (Messahel and Touili, 2024), no concrete example was provided. In this paper, we go one step further. We consider a case study and show *concretely* how this approach and this model can be applied to represent and analyse an example of a multi-threaded self modifying code infected with a malware.

1 INTRODUCTION

Self-modifying code is a programming philosophy that allows computer programs to change their behavior 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), others, use it to evade detection by anti-malware systems.

On the other hand concurrent programming is another technique of software engineering that allows to perform multiple tasks simultaneously to leverage up the use of the hardware resources seeking for more efficiency and performance improvement. Analyzing concurrent programs can be challenging due to the complex interactions and the inter-dependencies between concurrent threads and processes.

In this work, we consider the analysis problem of programs that present these two sources of difficulties: (1) concurrency and thread creation, and (2) self-modifying code. Analyzing this kind of code is challenging due to the complex features that it involves. Multiple techniques were introduced to analyse this kind of programs such as:

1. **Reverse engineering**: This technique involves analyzing the binary source code and manually understand and anticipate its behavior by using disassembling and decompiling tools, this approach can deliver certain results but it is hard and time consuming.
2. **Emulation** techniques which simulate the program's execution in a virtual environment using emulators. This can be used to observe the code's behaviour and detect malfunctioning.

However, these techniques have serious limitations. Indeed, reverse engineering is not automatic and necessitates human interaction, which can be very tedious. As for emulation techniques, they can analyse the program only *in a limited time interval*. To sidestep these difficulties, we introduced in (Messahel and Touili, 2024), a completely *automatic* and *static* approach to analyse *concurrent, self-modifying* code: we introduced 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 proposed in (Messahel and Touili, 2024) an algorithm to perform the backward reachability analysis of SM-DPNs. Our algorithm is based on (1) representing *infinite* sets of configurations of SMDPN using finite state automata, and (2) applying a kind of saturation procedure on these automata in

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

Messahel, W. and Touili, T.
Analyzing a Concurrent Self-Modifying Program: Application to Malware Detection.
DOI: 10.5220/0013103900003899

In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025) - Volume 1*, pages 176-182
ISBN: 978-989-758-735-1; ISSN: 2184-4356

order to apply in a backward way the different transitions of the SMDPN (these transitions correspond to the different instrcutions of the program).

However, the work of (Messahel and Touili, 2024) is too theoretic, and no running example was given to explain how this SMDPN model and this automata-based algorithm can be concretely applied to the analysis of real concurrent self-modifying code.

In this paper, we go one step further and consider a case study. We show *concretely* how the approach and the SMDPN model of (Messahel and Touili, 2024) can be applied to represent and analyse an example of a multi-threaded self modifying code infected with a malware.

2 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 malwares. A part of the community has used static techniques to analyze binary code, where they usually pre-process the binary code before conducting analysis (Schwartz et al., 2018; Chen et al., 2017; Zhang et al., 2018; Arzt et al., 2014; Biondo et al., 2018; Wu et al., 2019; Wang et al., 2017; Chen et al., 2017).

Analyzing self modifying code can be challenging due to the changing nature of the code. Some works use dynamic analysis approaches like (Dawei et al., 2018; Ugarte-Pedrero et al., 2015; Guizani et al., 2009; Bruschi et al., 2006; Anckaert et al., 2007; Blazy et al., 2016; Touili and Ye, 2019). However, none of these techniques can handle in a completely *automatic* way *concurrent* self-modifying code.

On the other hand, the ability to analyze concurrent and parallel programs is essential for understanding and improving the performance of these concurrent systems. Several works were proposed such as (Nethercote et al., 2007; Maisuradze et al., 2010; Liu et al., 2022; Alglave et al., 2010). However, none of these techniques consider concurrent *self-modifying* code.

The only work that we are aware of and that can handle *self-modifying and concurrent programs* is (Messahel and Touili, 2024). In this paper, we go one step further and show how the approach of (Messahel and Touili, 2024) can be applied in a concrete manner for the analysis of a self-modifying concurrent program infected with a malware.

3 MOTIVATING EXAMPLE

Executable instructions are stored in memory as byte code. Thus, changing the byte code will change the instruction itself. There are several kinds of self-modifying code. In this work, we consider self-modifying code caused by **self-modifying instructions**, which are often **mov** instructions that can access and modify the byte code stored in memory locations.

Consider the assembly code fragment shown in Listing 1. The program contains several self-modifying instructions that change its execution flow. It also contains a thread creation instruction introduced by the self-modifying instructions. You can see that the **mov** instruction was able to modify the instructions of the program successfully via its ability to read and write the memory. Let us explain step by step how this code contains thread creation and is self-modifying:

- Instruction **mov [0xb80201],0x2** will replace the content in the address **0xb80201** with **0x2**. Thus, the instruction **Mov eax,74e2h** at address **0xb80200** is replaced by **mov eax,0x2**.
- Instruction **mov [0xb80206],0xcd** will replace the content in the address **0xb80206** with **0xcd**, and instruction **mov [0xb80207],0x80** will replace the content in the address **0xb80207** with **0x80**. Thus, these two instructions will replace the instruction **Mov eax,eax** at address **0xb80206** with the instruction **Int 0x80**. This instruction will call the kernal with the parameter **0x2** in the **eax** register. This is a process creation function that will create a new child process that begins its execution right after the instruction **int 0x80**.

```
1  Address     Bytecode          Assembly
2
3  0xb80200    b8e2740000        Mov eax,74e2h
4  0xb80206    89c0              Mov eax,eax
5  0xb80208    83f800            Cmp eax,0x0
6  0xb8020b    7420              Jz child
7  0xb8020d    e821000000        Call func1
8  0xb80212    83f802            Cmp eax,0x2
9  0xb80215    e82a000000        Jz Exit
10 0xb8021a    c605010002b802    Mov [0xb80201],0x2
11 0xb80222    c605f7910408cd    Mov [0xb80206],0xcd
12 0xb80229    c605070002b880    Mov [0xb80207],0x80
13 0xb80230    e9ec6dfdaf        Jmp 0xb80200
```

Listing 1: Assembly code that contains self modifying instructions.

4 THE FORMAL MODEL: SELF-MODIFYING DYNAMIC PUSHDOWN NETWORK

We recall in this section the Self-Modifying Dynamic Pushdown Network model definition introduced in (Messahel and Touili, 2024).

A SM-PDS (Touili and Ye, 2017) is a push-down system that can modify its own set of rules during the execution. A Self modifying dynamic pushdown network (SM-DPN) consists of a network of SM-PDS (self modifying pushdown systems) that can model a network of pushdown processes running in parallel, where each of these pushdown systems can change its current set of rules and create new processes during its execution.

Formally, a SM-DPN (Messahel and Touili, 2024) is a tuple $\mathfrak{R} = (P, \Gamma, \Delta, \Delta_c)$, where P is a finite set of control points, Γ is a finite set of stack symbols, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*) \cup (P \times \Gamma) \times (P \times \Gamma^*) \times ((P \times 2^{\Delta \cup \Delta_c}) \times \Gamma^*)$, $\Delta_c \subseteq P \times (\Delta \cup \Delta_c) \times (\Delta \cup \Delta_c) \times P$ is a finite set of modifying transitions rules. A DPN is a SM-DPN with $\Delta_c = \emptyset$.

Each process in the network has its current set of transition rules θ called the phase, such as $\theta \subseteq \Delta \cup \Delta_c$, rules in Δ_c can change a process phase. There are 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 expresses 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 expresses 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 expresses 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. As in (Messahel and Touili, 2024), to simplify the presentation, 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 expresses that there are n running processes in the network and for every i such as $0 \leq i \leq n$, 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 $Conf_\mathfrak{R}$ be the set of all global configurations of the SM-DPN $\mathfrak{R}$. We define the transition relation $\Rightarrow_\mathfrak{R}$ to be the smallest relation between two configurations in $Conf_\mathfrak{R} \times Conf_\mathfrak{R}$ as follows:

- Let $c = up^\theta wv$, $c' = up'^{\theta'} wv$ with $u, v \in Conf_\mathfrak{R}$, $w \in \Gamma^*$, if $r = p \xrightarrow{(r_1,r_2)} p' \in \Delta_c \cap \theta$, $r_1 \in \theta$ and $\theta' = (\theta \setminus \{r_1\}) \cup \{r_2\}$ then c is a predecessor of c' (also written as $c \Rightarrow_\mathfrak{R} c'$). 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 $\gamma \in \Gamma$, $w, w' \in \Gamma^*$, $u, v \in Conf_\mathfrak{R}$, and $r = p\gamma \hookrightarrow p'w' \in \Delta \cap \theta$, then $c \Rightarrow_\mathfrak{R} c'$. 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 $\gamma \in \Gamma$, $w \in \Gamma^*$, $u, v \in Conf_\mathfrak{R}$, if $r = p\gamma \hookrightarrow p'w' \triangleright p_1^{\theta_1} w_1 \in \Delta \cap \theta$, then $c \Rightarrow_\mathfrak{R} c'$. Here the rule r will move the SM-DPN process from the control point p to the control point p', pops γ from the stack, pushes w' into the stack and creates 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 $\Rightarrow^*_\mathfrak{R}$ as the transitive reflexive closure of $\Rightarrow_\mathfrak{R}$. If a configuration c' is reachable from c_0 in i steps by applying $\Rightarrow_\mathfrak{R}$ i times, we write $c_0 \Rightarrow^i_\mathfrak{R} c'$.

We denote the set of immediate predecessors (resp. successors) of a configuration c as $Pre_\mathfrak{R}(c) = \{c_1 \in Conf_\mathfrak{R} : c_1 \Rightarrow_\mathfrak{R} c\}$ (resp. $Post_\mathfrak{R}(c) = \{c_1 \in Conf_\mathfrak{R} : c \Rightarrow_\mathfrak{R} c_1\}$). Let $Pre^*_\mathfrak{R}$(resp. $Post^*_\mathfrak{R}$) denote the reflexive transitive closure of $Pre_\mathfrak{R}$ (resp. $Post_\mathfrak{R}$).

These notations can be generalized to sets of configurations in the obvious ways. We omit the subscript $\mathfrak{R}$ if clear from the context.

5 MODELING SELF MODIFYING CONCURRENT CODE WITH SM-DPN

To perform binary code analysis, we need a proper intermediate representation that can be easily converted to SM-DPN. For this purpose, we use CFGs (Control Flow Graphs) automatically extracted from the binary code. Then, extracting a SM-DPN from a binary program is segmented to three main steps:

1. **Generating the Assembly Code:** Firstly, we disassemble the binary code with a proper tool that reverses engineer the executed binary code to assembly instructions. For example, **83 f8 00** will be converted to **Cmp eax,0x0**.
2. **Converting the Extracted Assembly Code to a CFG:** Secondly, the previous step will provide us with a list of assembly instructions alongside their memory addresses, from here we can easily extract an abstracted version of the code in the form of a control flow graph.
3. **Parsing the CFG to SM-DPN rules:** Lastly, we will convert the extracted CFG to a SM-DPN that models dynamic process creation and self modifying instructions. We suppose we are given an oracle that computes the values (or their approximations) of the different registers at different control points of the program. Let Γ be the set of all memory adresses and register values of the program. There are four possible cases depending on the CFG instructions:
   - If the CFG instruction is of the form
     $$n_1 \xrightarrow{push\beta} n_2$$
     it will be converted to SM-DPN rules of the form
     $$n_1\gamma \hookrightarrow n_2\beta\gamma$$
     for every γ in Γ.
   - If the CFG instruction is of the form
     $$n_1 \xrightarrow{pop\beta} n_2$$
     it will be converted to SM-DPN rules of the form
     $$n_1\beta \hookrightarrow n_2\varepsilon$$
   - If the CFG instruction is of the form
     $$n_1 \xrightarrow{call\ f} n_2$$
     it will be converted to SM-DPN rules of the form
     $$n_1\gamma \hookrightarrow n_f n_2\gamma$$
     for every γ in Γ, where n_f is the entry point of the procedure f.
   - If the CFG instruction is of the form
     $$n_0 \xrightarrow{call\,CreateProcess} n_1$$
     where CreateProcess is a function that creates a new process, then it will be converted to SM-DPN rules of the form
     $$n_0\gamma \hookrightarrow n_1\ \gamma \triangleright p_0 w_0$$
     for every γ in Γ, where p_0 is the entry point of the created process and w_0 is its initial stack content.
   - If the CFG instruction is of the form
     $$n_1 \xrightarrow{mov\,m,l} n_2$$
     where m is an executable code address, then this is a self modifying code instruction. This instruction will be converted to SM-DPN rules of the form
     $$n_1 \overset{r_1,r_2}{\hookrightarrow} n_2$$
     where r_1 is the rule that corresponds to the instruction being modified by $n_1 \xrightarrow{mov\,m,l} n_2$, and r_2 is the rule corresponding to the new instruction that replaces the first one.

6 SM-DPN BACKWARD REACHABILITY ANALYSIS

It has been shown in (Messahel and Touili, 2024) that if a regular set of global configurations can be represented by a special kind of finite automata, then it is possible to effectively compute a finite automaton that represents the set of all backward reachable configurations. We recall in this section the results of (Messahel and Touili, 2024) concerning the backward reachability analysis of SM-DPNs.

Let $\mathfrak{R} = (P, \Gamma, \Delta, \Delta_c)$ be a SM-DPN. To finitely represent a regular infinite set of SM-DPN configurations, we use a special kind of automata: a $\mathfrak{R}$-automaton (Messahel and Touili, 2024) 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 $T' \subseteq S_s \times \Gamma \times (S_s \backslash \{s_{p^\theta} : s \in S_c,\ p \in P,\ \theta \subseteq \Delta \cup \Delta_c\}) \cup S_s \times \{\varepsilon\} \times S_c$ 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 $\gamma \in \Gamma \cup \{\varepsilon\}$ and $s, s' \in S$, if $(s, \gamma, s') \in T$ we write $s \xrightarrow{\gamma}_T s'$. This notation can be extended in the obvious manner to sequences of symbols as follows :

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

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

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

It was shown in (Messahel and Touili, 2024) that :

Theorem 6.1. *Let $\mathfrak{R} = (P, \Gamma, \Delta, \Delta_c)$ be a SM-DPN and $A = (S, \Omega, T, s^0, F)$ be an $\mathfrak{R}$-automaton. We can build a finite automaton $A_{pre^*} = (S, \Omega, T', s^0, F)$ such that $L(A_{pre^*}) = pre^*(L(A))$.*

7 CASE OF STUDY: A REAL EXAMPLE

In order to observe the workflow of a binary analysis using SM-DPN modeling and reachability analysis, let us consider an assembly code extracted from a binary file infected by a concurrent self modifying variation of 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 (Perekalin, 2017). The binary is obtained from **MalwareCollection** github repo (Enderman, 2022). We explain step by step the process of reverse engineering the binary "BadRabbit.exe", then modeling it with a SM-DPN, to finally apply the Pre^* algorithm of (Messahel and Touili, 2024) to perform a reachability analysis where the malicious part starts from the address n_7.

7.1 Reverse Engineering

We use radare2 (Studer et al., 2023) to decompile the binary and retrieve its assembly code showed in Listing 2, where n_0, n_1, ..., n_i are memory addresses.

```
1  Address        Assembly
2
3  n_0            Push 0x10
4  n_1            Mov ebp, 0x1ff
5  n_2            Mov [n_12], 0xffd6 ; change
       instruction stored in n_12
6  n_3            Nop
7  n_4            Push eax
8  n_5            Mov [var_1250h], 0
9  n_6            jmp n_2
10 n_7            LEA EDX,[0xfffff9e4]
11 n_8            PUSH EDX
12 n_9            MOV [local_12b0],0x44
13 n_10           CALL CreateProcessW; create new
       process
```

Listing 2: BadRabbit snippet ASM code.

7.2 Translating the Binary Code to an SM-DPN

To model the assembly code of the BadRabbit malware with a SM-DPN, we loop over the instructions and convert each instruction to a SM-DPN rule as described below. In order to convert some instructions, we need to resolve its registers values. To this end, we set up three levels of precision:

1. using Radare2 (Studer et al., 2023) built-in. This is the fastest method but it can only resolve the IP register.
2. using Angr (Shoshitaishvili et al., 2016) Symbolic execution. This performs a symbolic exe-

cution without actually running the program and can track register values in most of the cases, but it is costly in terms of time and memory consumption because it is based on exploring all possible logical branches.

3. by running the program in an isolated mode with a debugger. This is the best method in terms of time and memory consumption, but it is tricky if the program depends on a variety of external inputs.

Following the rules that translate an assembly code to a SM-DPN described in section 5, we translate each instruction to SM-DPN rules as follows, where $p_0, p_1, \ldots, p_i$ represent the SM-DPN control points according to the addresses n_0, n_1, ..., n_i, respectively, such that p_0 is the program control point at the address n_0. Let Γ be the set of all memory adresses and register values of the program. The instructions of the program are translated to SM-DPN rules as follows:

- push 0x10 : since this is a push instruction, we move from control point p_0 to control point p_1 and we push 0x10 to the stack. Thus, this instruction can be modelled by these SM-DPN rules: $p_0\gamma \hookrightarrow p_1$ 0x10γ for every $\gamma \in \Gamma$.
- mov ebp, 0x1ff : a non self modifying mov instruction does not change the content of the stack and moves the SM-DPN from control point p_1 to control point p_2 resulting in rules $p_1\ \gamma \hookrightarrow p_2\ \gamma$, for every $\gamma \in \Gamma$.
- mov [n_12], 0xffd6 : a self modifying mov instruction does not change the content of the stack, moves the SM-DPN from control point p_2 to control point p_3 and replaces an instruction represented by a rule, say r', by another instruction represented by a rule, say r", resulting in the SM-DPN rule: $p_2 \xrightarrow{(r',r")} p_3$.
- Nop : this instruction moves the SM-DPN from control point p_3 to control point p_4 resulting in rules $p_3\ \gamma \hookrightarrow p_4\ \gamma$, for every $\gamma \in \Gamma$.
- push eax : a push instruction moves from control point p_4 to control point p_5 and pushes the value of eax to the stack. Thus, this instruction is translated to the following SM-DPN rules: $p_4\ \gamma \hookrightarrow p_5eax\gamma$, for every $\gamma \in \Gamma$.
- mov [var_1250h], 0 : a non self modifying mov instruction does not change the content of the stack and moves the SM-DPN from control point p_5 to control point p_6, resulting in the SM-DPN rules $p_5\gamma \hookrightarrow p_6\gamma$, for every $\gamma \in \Gamma$.
- jmp n_2 : The jmp instruction does not change the stack content and moves the SM-DPN from control point p_6 to control point p_2 resulting in the SM-DPN rules: $p_6\gamma \hookrightarrow p_2\gamma$, for every $\gamma \in \Gamma$.
- push edx : a push instruction moves the SM-DPN from control point p_8 to control point p_9 and pushes edx to the stack. Thus, this instruction can be translated to the following SM-DPN rules: $p_8\gamma \hookrightarrow p_9edx\gamma$, for every $\gamma \in \Gamma$.
- mov [local_12b0],0x44 : a non self modifying mov instruction does not change the stack content and moves the SM-DPN from the control point p_9 to control point p_{10} resulting in the SM-DPN rules: $p_9\ \gamma \hookrightarrow p_{10}\ \gamma$, for every $\gamma \in \Gamma$.
- call CreateProcessW : a thread creation instruction moves the SM-DPN from control point p_{10} to control point p_{11}, while launching a new thread. Suppose, the function CreateProcessW creates a new process having s_0 as entry control point and w_0 as stack content, then, this instruction is translated to the following SM-DPN rules: $p_{10}\ \gamma \hookrightarrow p_{11}\ \gamma \triangleright s_0\ w_0$, for every $\gamma \in \Gamma$.

7.3 The BadRabbit Reachability Analysis

Once we get the above SM-DPN model, we are ready to apply the backward reachability results of Theorem 6.1 to analyse the above program. These techniques, that were implemented in a tool to analyze SM-DPNs, have found that the program entry point (n_0) is in the pre^* of the malicious entry point (n_7), which means that the program is infected.

8 CONCLUSION

In this paper, we tackle the analysis problem of multi-threaded parallel programs that contain self modifying code, i.e., code that can modify itself during the execution time. Malware use heavily this kind of self-modifying code in order to get obfuscated so that they cannot be detected by anti-viruses. To model such programs, we use a new model called Self Modifying Dynamic Pushdown Network (SM-DPN). 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. To analyse self modifying concurrent programs, we perform reachability analysis of SM-DPNs. We successfully apply our approach to represent and analyse a multi-threaded self modifying code infected with a malware.

REFERENCES

Alglave, J., Kashyap, M., and Tofte, M. (2010). Compositional reasoning for shared-variable concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(5):1–53.

Anckaert, B., Madou, M., and De Bosschere, K. (2007). A model for self-modifying code. In Camenisch, J. L., Collberg, C. S., Johnson, N. F., and Sallee, P., editors, *Information Hiding*, pages 232–248, Berlin, Heidelberg. Springer Berlin Heidelberg.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269.

Biondo, A., Conti, M., and Lain, D. (2018). Back to the epilogue: Evading control flow guard via unaligned targets. In *Ndss*.

Blazy, S., Laporte, V., and Pichardie, D. (2016). Verified abstract interpretation techniques for disassembling low-level self-modifying code. *Journal of Automated Reasoning*, 56:283–308.

Bruschi, D., Martignoni, L., and Monga, M. (2006). Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings 3*, pages 129–143. Springer.

Chen, Y., Zhang, D., Wang, R., Qiao, R., Azab, A. M., Lu, L., Vijayakumar, H., and Shen, W. (2017). Norax: Enabling execute-only memory for cots binaries on aarch64. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 304–319. IEEE.

Dawei, S., Delong, L., and Zhibin, Y. (2018). Dynamic self-modifying code detection based on backward analysis. In *Proceedings of the 2018 10th International Conference on Computer and Automation Engineering*, ICCAE 2018, page 199–204, New York, NY, USA. Association for Computing Machinery.

Enderman (2022). Malwarecollection. https://github.com/xcp3r/MalwareCollection.

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

Liu, Y., Xu, Z., Fan, M., Hao, Y., Chen, K., Chen, H., Cai, Y., Yang, Z., and Liu, T. (2022). Concspectre: Be aware of forthcoming malware hidden in concurrent programs. *IEEE Transactions on Reliability*, 71:1–10.

Maisuradze, G., Petrenko, A. S., Bala, A., and Lie, D. (2010). Threadsanitizer: finding data races in native code. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100.

Messahel, W. and Touili, T. (2024). Reachability analysis of concurrent self-modifying code. In *28th International Conference on Engineering of Complex Computer Systems (ICECCS)*.

Nethercote, N., Seward, J., and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 International Symposium on Dynamic Languages*, pages 89–100.

Perekalin, A. (2017). Bad rabbit: A new ransomware epidemic is on the rise. https://www.kaspersky.com/blog/bad-rabbit-ransomware/19887/.

Schwartz, E. J., Cohen, C. F., Duggan, M., Gennari, J., Havrilla, J. S., and Hines, C. (2018). Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 426–441.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. (2016). SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

Studer, A., Abd El-MAwgood, A. M., and Akshay Krishnan, R. (2023). The official radare2 book. https://book.rada.re/credits/credits.html.

Touili, T. and Ye, X. (2017). Reachability analysis of self modifying code. In *22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*.

Touili, T. and Ye, X. (2019). Ltl model checking of self modifying code. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*.

Ugarte-Pedrero, X., Balzarotti, D., Santos, I., and Bringas, P. G. (2015). Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673.

Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., and Vigna, G. (2017). Ramblr: Making reassembly great again. In *NDSS*.

Wu, W., Chen, Y., Xing, X., and Zou, W. (2019). Kepler: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *USENIX Security Symposium*, pages 1187–1204.

Zhang, X., Zhang, Y., Mo, Q., Xia, H., Yang, Z., Yang, M., Wang, X., Lu, L., and Duan, H. (2018). An empirical study of web resource manipulation in real-world mobile applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1183–1198.