LTL Model Checking of Concurrent Self Modifying Code*

Tayssir Touili and Olzhas Zhangeldinov

IRIF, CNRS, and University Paris Cité

Abstract. We consider the LTL model-checking problem of concurrent self modifying code, i.e., concurrent code that has the ability to modify its own instructions during execution time. This style of code is frequently utilized by malware developers to make their malicious code hard to detect. To model such programs, we consider Self-Modifying Dynamic Pushdown Networks (SM-DPN). A SM-DPN is a network of Self-Modifying Pushdown processes, where each process has the ability to modify its current set of rules and to spawn new processes during execution time. We consider model checking SM-DPNs against single indexed LTL formulas, i.e., conjunctions of separate LTL formulas on each single process. This problem is non trivial since the number of spawned processes in a given run can be infinite. Our approach is based on computing finite automata representing the set of configurations from which the SM-DPN has a run that satisfies the single-indexed LTL formula. We implemented our techniques in a tool and obtained promising results. In particular, our tool was able to detect a concurrent, self-modifying malware.

1 Introduction

Most of the programs implement concurrent routines for efficiency. However, analysis of concurrent programs is a notoriously hard challenge. Therefore, significant efforts were made in the direction of automatic verification of concurrent programs [4, 16, 20, 25].

On the other hand, self-modifying code is a code that modifies its own instructions during the rexecution time. This technique is widely used by packers to decrease the size of a program and by malware developers to confuse antivirus software and make their malware hard to detect. The problem of analysing self-modifying code was approached by more recent studies [6, 9, 20, 26].

This paper focuses on analysing programs that are both *concurrent* and *self-modifying*. Indeed, modern malware, e.g. some variants of Mirai, employs concurrency for parallel execution of different tasks and contains self-modifying code to stay undetected for as long as possible.

Self-modifying behaviour of a program is achieved by writing to the executable region of the binary, which is an array of memory locations from where

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

```
# Address   Bytecode            Assembly
  0x04       0x31c0              xor     eax, eax
  0x06       0xb001              mov     al, 1
  0x08       0xbb80cd02b0        mov     ebx, 0xb002cd80
  0x0d       0x891d04000000      mov     [0x04], ebx
  0x13       0xebe7              jmp     0x04
```

Listing 1.1. Binary code with a self-modifying instruction.

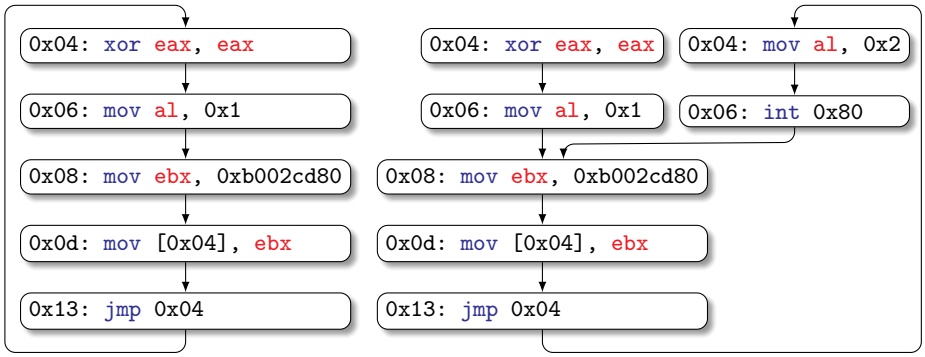
```
# Address   Bytecode            Assembly
  0x04       0xb002              mov     al, 0x2
  0x06       0xcd80              int     0x80
  0x08       0xbb80cd02b0        mov     ebx, 0xb002cd80
  0x0d       0x891d04000000      mov     [0x04], ebx
  0x13       0xebe7              jmp     0x04
```

Listing 1.2. Binary code with after executing a self-modifying instruction.

a computer reads instructions to execute. Self-modifying behaviour can be implemented using different techniques. For example, let us consider a self-modifying concurrent binary code of a Linux program running on a CPU with x86 architecture. Programs for this architecture mostly use `mov` instructions to write data into memory, including the memory of executable instructions. A portion of the program's assembly is demonstrated in Listing 1.1. The first column denotes relative addresses of instructions. The second column contains bytes stored at that address, and the third column is the corresponding assembly code for the binary code. `eax` and `ebx` are CPU registers, and `al` points to the lowest byte of `eax`. For example, two bytes stored at the memory address `0x04` are `0x31c0`, which is a bytecode for the assembly instruction `xor eax, eax`, which sets `eax` to 0.

Let us explain why this code is self-modifying and concurrent. First, a process starts executing the program at the address `0x04` and reads bytes `0x31c0` stored at this location. Bytecode `0x31c0` corresponds to the instruction `xor eax, eax`, which means the process sets `eax` to 0. This instruction is two bytes long, so the process reads the next instruction from the address `0x06`. This address contains bytes `0xb001`, which is the bytecode of the instruction `mov al, 0x1`, which sets the lowest byte of `eax` to `0x1`. Then, the process executes the instruction `mov ebx, 0xb002cd80`, which corresponds to the bytecode `0xbb80cd02b0` stored at the address `0x08`. This sets `ebx` to `0xb002cd80`. Next, the process executes the instruction stored at the address `0x0d`. This instruction is `mov [0x04], ebx` and it stores the values of `ebx` (previously set to `0xb002cd80`) to the address `0x04`. This changes the instructions stored at address `0x04`. Since `0xb002cd80` is the binary code for instructions `mov al, 0x2` and `int, 0x80`, the instruction at address `0x04` and `0x06` will be replaced by these two instructions. Therefore, `mov [0x04], ebx` is a self-modifying instruction. The code of the program after self-modification occurs is presented in Listing 1.2. The next instruction the process reads will be `jmp 0x04` (corresponding to the bytecode `0xebe7` contained at

the location `0x13`), which will make the process jump to the address `0x04`. As explained, this address now contains a modified instruction with the bytecode `0xcd80`. The new assembly is `mov al, 0x2`, which sets `eax` to 2 (remember that we set higher bytes of `eax` to 0 with `xor eax, eax`). Then, the process will execute the modified instruction at the address `0x06` with the new bytecode `0xcd80`, which corresponds to the assembly `int, 0x80`. This instruction tells Linux kernel to execute a system function (or system call / syscall) with the function code stored in `eax`. The function code `0x2` corresponds to the kernel's *fork* function, which spawns a copy of the current process. Since the previous instruction sets `eax` to `0x2`, Linux kernel executes the *fork* function, making this program also concurrent.



(a) Naive CFG (b) Accurate CFG

Fig. 1. Naive and accurate CFGs of code from Listing 1.1.

You can see that if we analyze this program blindly, using the instructions of Listing 1.1, without taking into account the self-modifying nature of instruction `mov [0x04], ebx`, then we will obtain the Control Flow Graph (CFG) as shown in Figure 1(a). However in reality, the program will spawn parallel processes indefinetely. This is clear if we look at the more accurate CFG in Figure 1(b). This is one of the tecnhiques malware developers use to obscure the real behaviour of the malware from antivirus software. Therefore, it is necessary to take into account the self-modifying nature of the code for an accurate analysis.

The aim of this paper is to provide an efficient algorithm for model checking such self-modifying and concurrent code. To analyse such kind of programs, we need an abstract model suitable for both self-modifications and concurrency. Pushdown System (PDS) is a natural abstraction for sequential programs that utilize stack and can call recursive functions [14]. Since modelling sequential execution is not enough for concurrent programs, Dynamic Pushdown Networks (DPNs) were proposed to model how a program can spawn parallel processes [8]. A DPN is a network of pushdown systems with each PDS being able to spawn a new process controlled by another PDS. On the other hand, to model pro-

grams with self-modifying instructions, authors of [26] proposed Self-Modifying Pushdown Systems (SM-PDS). Intuitively, SM-PDS is a PDS with the ability to modify its set of rules during runtime. To model both concurrent and self-modifying programs, a previous work [20] introduced Self-Modifying Dynamic Pushdown Networks (SM-DPNs) as a network of self-modifying pushdown systems. [20] proposed an efficient algorithm for reachability analysis of SM-DPNs. In this work, we go one step further and propose an efficient LTL model checking algorithm for SM-DPNs.

Model checking concurrent programs imposes additional challenges. In fact, model checking LTL formulas that reason about two concurrent processes, i.e. formulas where atomic propositions are distributed over the control states of two processes, is undecidable [17], even in the absence of thread creation. To overcome this problem, we consider single-indexed LTL formulas of the form $f = \bigwedge_i f_i$, such that f_i is an LTL formula over process i. This problem of single-indexed LTL model checking of DPNs was tackled by [25], however, this work does not take into account self-modification of programs.

In this paper, we go one step further and consider LTL model checking of Self-Modifying Dynamic Pushdown Networks (SM-DPNs). Since SM-DPNs are equivalent to standard DPNs, we could translate the SM-DPN into a DPN, and then use the LTL model checking algorithm of [25]. But, as shown by the experiments in Section 7, this approach is not efficient. Therefore, we propose a *direct* and *efficient* model checking algorithm for SM-DPNs against single-indexed LTL formulas. First, we tackle the problem of LTL model checking of a single process. We construct an automaton $\mathcal{A}_i$ for each sequential process i, such that $\mathcal{A}_i$ accepts a configuration of the process i if it has a run that satisfies the corresponding LTL formula f_i. During the construction of $\mathcal{A}_i$, to tackle the self-modifying instructions, we keep track of the current *phase* of the process i(the current set of the transitions of the process). $\mathcal{A}_i$ also keeps track of the spawned processes during the execution because we need to check that every spawned process j satisfies the formula f_j as well. Then, we use all of the obtained automata to compute the largest set of processes $\mathcal{D}_{fp}$, such that every process i in $\mathcal{D}_{fp}$ satisfies the LTL formula f_i, and does not spawn a process j that violates the formula f_j ($j \geq 0$). Then, we check that every initial process satisfies the LTL formula and spawns only processes from $\mathcal{D}_{fp}$. Our experiments show that our direct approach is much more efficient than model checking an equivalent DPN using the approach in [25]. Moreover, we show the applicability of our approach for malware detection.

Related Works. Model checking of sequential binaries has been extensively studied in [17, 24]. However, these studies do not consider neither concurrency of programs, nor self-modifying instructions.

To solve the problem of model checking concurrent programs, different models were proposed. Some studies use the Dynamic Pushdown Network (DPN) model [8,21,25] and its extensions [12]. Other studies [18,22] performed model checking on networks of pushdown systems. However, these works do not consider self-modifying code.

To analyze self-modifying programs, several dynamic analysis approaches were proposed [11,27], which imply executing the binary in a debugger and observe the behaviour of the program. However, these techniques do not allow to analyze every possible behaviours of the program. Static analysis of self-modifying code was proposed in, for example, [6,9]. However, [9] needs extra invariant annotations. As for [6], it proposes an abstract representation without a specific approach to automated analysis. Another model to represent self-modifying code is State-Enhanced Control Flow Graph (SE-CFG) [3]. Reachability analysis of binaries with self-modifying instructions was also proposed by [5]. However, both of these studies [3,5] do not take into account the stack of the program, and thus, do not provide an accurate enough model of execution. Self-modifying pushdown systems (SM-PDS) were successfully used for model checking self-modifying programs [26]. However, these works do not support concurrency.

As far as we know, the only work that considers both concurrent and self-modifying programs is [20], where the SM-DPN model was proposed. But, [20] considers only reachability analysis. In this paper, we go one step further and propose an efficient LTL model checking of SM-DPNs.

Outline. Section 2 introduces our SM-DPN model. Section 3 proposes our algorithm for model checking SM-DPNs. Section 4 describes how the LTL model checking of one process can be reduced to the computation of predecessors. Section 5 proposes an efficient automata-based approach for computing the regular set of predecessors of a single process. Section 7 describes the practical experiments conducted using the proposed LTL model checking. For lack of space, proofs are omitted. They can be found in the full version of the paper.

2 Preliminaries

In this section, we introduce Self-Modifying Pushdown Networks (SM-DPNs).

2.1 Self-Modifying Pushdown Network

A *Self-Modifying Dynamic Pushdown Network* (SM-DPN) is an extension of standard Pushdown Systems (PDS) that models programs that can spawn parallel processes and can change their instruction sets in real-time. SM-DPN consists of several *Self-Modifying Dynamic Pushdown Systems* (SM-DPDS) each modelling a single sequential process. Formally:

Definition 1. *A Self-Modifying Dynamic Pushdown Network (SM-DPN) is a tuple* $\mathcal{M} = (\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$*, such that for every* i*,* $1 \leq i \leq n$*,* $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i, \Delta_i^c)$ *is a Self-Modifying Dynamic Pushdown System (SM-DPDS), where* P_i *is a finite set of control locations (for any* $j \neq k$*,* $P_j \cap P_k = \emptyset$*),* Γ_i *is the stack alphabet,* Δ_i *is a finite set of rules of the forms: (a)* $p\gamma \hookrightarrow p_1\omega_1$ *and (b)* $p\gamma \hookrightarrow p_1\omega_1 \triangleright p_2\omega_2\theta_2$*, such that* $p, p_1 \in P_i$*,* $\gamma \in \Gamma_i$*,* $\omega_1 \in \Gamma_i^*$*,* $p_2 \in P_j$*,* $\omega_2 \in \Gamma_j^*$*,* $\theta_2 \subseteq \Delta_j \cup \Delta_j^c$*, and* Δ_i^c *is a finite set of self-modifying rules of the form* $p \xhookrightarrow{(\rho_1,\rho_2)} p_1$*, such that* $\rho_1, \rho_2 \subseteq \Delta_i \cup \Delta_i^c$*. A Dynamic Pushdown System (DPDS) is a SM-DPDS, such*

that $\Delta_i^c = \emptyset$. *A Dynamic Pushdown Network (DPN) is a SM-DPN, such that for every* $1 \leq i \leq n$, $\mathcal{P}_i$ *is a DPDS.*

Consider a SM-DPN $\mathcal{M} = (\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$. A SM-DPDS $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i, \Delta_i^c)$ can be seen as a Pushdown System with the ability (1) to spawn a new process and (2) to change its current set of rules during its execution. Because a process modelled by a SM-DPDS can change its set of rules at runtime, we introudce the notion of a *phase*. A phase $\theta \subseteq \Delta_i \cup \Delta_i^c$ is the current set of rules that can be applied. θ changes when a self-modifying rule of type $p \xhookrightarrow{(\rho_1, \rho_2)} p_1 \in \theta$ is applied. Such rule denotes that if the process is at the control location p, it can transition to the control location p_1 while removing rules in ρ_1 from θ and adding rules from ρ_2 to θ. Note that unlike the definition of SM-DPN in [20], we allow ρ_1 and ρ_2 have different numbers of rules. The rules of type $p\gamma \hookrightarrow p_1\omega_1 \in \theta$ define that if the process is at the control location p and has γ on the top of its stack, then it can pop γ from the stack, push ω_1 onto it, and go to the control location p_1. Similarly, the rules of type $p\gamma \hookrightarrow p_1\omega_1 \triangleright p_2\omega_2\theta_2 \in \theta$ describe the same behaviour as $p\gamma \hookrightarrow p_1\omega_1$ but additionally, the rule spawns another process at the control location p_2, with ω_2 as content of the stack, and phase θ_2. This new process will be executed by its corresponding SM-DPDS $\mathcal{P}_j = (P_j, \Gamma_j, \Delta_j, \Delta_j^c) \in \mathcal{M}$, such that $p_2 \in P_j$.

2.2 Configurations and DCLICs

A *local configuration* of a SM-DPDS $\mathcal{P}_i$ is a tuple $(\langle p_i, \omega_i \rangle, \theta_i)$, where $p_i \in P_i$ is the current state of the process, $\omega_i \in \Gamma_i^*$ is the current stack content, and $\theta_i \subseteq \Delta_i \cup \Delta_i^c$ is the current phase. The set of all local configurations of a process of $\mathcal{P}_i$ is denoted as $Conf_i$. A *global configuration* of a SM-DPN is a multi-set over $\bigcup_{i=1}^{n} Conf_i$.

When a process can spawn a new process with a local configuration $(\langle p_j, \omega_j \rangle, \theta_j)$, we say that $p_j\omega_j\theta_j$ is a *Dynamically Created Local Initial Configuration* (DCLIC). The finite set of all DCLICs created by a process of $\mathcal{P}_i$ is denoted as $\mathcal{D}_i$ and is equal to $\{p_2\omega_2\theta_2 \in P_j \times \Gamma_j^* \times 2^{\Delta_j \cup \Delta_j^c} | \exists p, p' \in P_i, \gamma \in \Gamma_i, \omega' \in \Gamma_i^*, p\gamma \hookrightarrow p'\omega' \triangleright p_2\omega_2\theta_2 \in \Delta_i\}$.

For a SM-DPDS $\mathcal{P}_i$ and a set of DCLICs $D \subseteq \mathcal{D}_i$, we define the *successor operator* $\overset{D}{\Longrightarrow}_i$ on a pair of local configurations of $\mathcal{P}_i$ as follows: $(\langle p, \omega \rangle, \theta) \overset{D}{\Longrightarrow}_i (\langle p', \omega' \rangle, \theta')$ means that a process at the configuration $(\langle p, \omega \rangle, \theta)$ can transition into the configuration $(\langle p', \omega' \rangle, \theta')$ by applying one of the rules in the current phase θ and the rule applied spawns processes with DCLICs D. Formally, $(\langle p, \omega \rangle, \theta) \overset{D}{\Longrightarrow}_i (\langle p', \omega' \rangle, \theta')$ iff one of these conditions holds:

1. $\exists \gamma \in \Gamma_i, u, v \in \Gamma_i^*$, s.t. $\omega = \gamma u$, $\omega' = vu$, $\theta = \theta'$, $D = \emptyset$, and $p\gamma \hookrightarrow p'v \in \theta$, or
2. $\omega = \gamma u$, $\omega' = vu$, $\theta = \theta'$, $D = \{p_2\omega_2\theta_2\}$, and $p\gamma \hookrightarrow p'v \triangleright p_2\omega_2\theta_2 \in \theta$, or
3. $\omega = \omega'$, $D = \emptyset$, $p \xhookrightarrow{(\rho_1, \rho_2)} p' \in \theta$, $\theta' = (\theta \backslash \rho_1) \cup \rho_2$.

Intuitively, condition 1 specifies that if the process is at the local configuration $(\langle p, \gamma u\rangle, \theta)$, such that the rule $p\gamma \hookrightarrow p'v$ is in the current phase θ, then the process can pop γ from the stack, push v onto it, and transition to the state p' without spawning any new process, getting $(\langle p', vu\rangle, \theta)$. Condition 2 means that if the process is at the local configuration $(\langle p, \gamma u\rangle, \theta)$, such that the current phase θ contains a rule $p\gamma \hookrightarrow p'v \triangleright p_2\omega_2\theta_2$, then it can pop γ from the top of the stack, push v, go to p', and spawn a new process with the DCLIC $p_2\omega_2\theta_2$, i.e. starting at local configuration $(\langle p_2, \omega_2\rangle, \theta_2)$, getting $(\langle p', vu\rangle, \theta)$. Condition 3 defines that if the process is at the local configuration $(\langle p, u\rangle, \theta)$, such that the rule $p \xhookrightarrow{(\rho_1,\rho_2)} p'$ is in the current phase θ, the process can remove all rules $r_1 \in \rho_1$ from the phase θ and add all rules $r_2 \in \rho_2$ to θ, changing the current state to p', getting $(\langle p', u\rangle, \theta')$, such that $\theta' = (\theta \backslash \rho_1) \cup \rho_2$.

For local configurations $c, c' \in Conf_i$ and set of DCLICs $D \subseteq \mathcal{D}_i$, we define a reflexive-transitive closure $c \overset{D}{\Longrightarrow}_i^* c'$ as follows, where $c'' \in Conf_i$ and $D', D'' \subseteq \mathcal{D}_i$: (1) $c \overset{\emptyset}{\Longrightarrow}_i^* c$ and (2) if $c \overset{D'}{\Longrightarrow}_i c'$ and $c' \overset{D''}{\Longrightarrow}_i^* c''$, then $c \overset{D}{\Longrightarrow}_i^* c''$, where $D = D' \cup D''$. We also define the non-reflexive transitive closure $c \overset{D}{\Longrightarrow}_i^+ c'$ as follows: $c \overset{D}{\Longrightarrow}_i^+ c'$ iff $\exists D', D'' \subseteq \mathcal{D}_i$, such that $D = D' \cup D''$, $c \overset{D'}{\Longrightarrow}_i c''$ and $c'' \overset{D''}{\Longrightarrow}_i^* c'$.

Consider an arbitrary set of pairs of local configurations and sets of DCLICs $W \subseteq Conf_i \times 2^{\mathcal{D}_i}$. Let $pre : 2^{Conf_i \times 2^{\mathcal{D}_i}} \to 2^{Conf_i \times 2^{\mathcal{D}_i}}$ be such that $pre(W) = \{(c, D \cup D'), c \in Conf_i | \exists D \subseteq \mathcal{D}_i, (c', D') \in W : c \overset{D}{\Longrightarrow}_i c'\}$. Let pre^+ and pre^* be the transitive and reflexive-transitive closures of pre, respectively. In other words, pre takes a pair of a local configuration and a set of DCLICs, and returns a set of predecessors of the given configurations paired with a superset of the given DCLICs that will be generated by the predecessors. Consider a local configuration c' and a set of DCLICs D'.

A *local run* of $\mathcal{P}_i$ is a possibly infinite sequence of local configurations $c_0c_1c_2\ldots$, s.t. $\forall j \geq 0 : \exists D \subseteq \mathcal{D}_i : c_j \overset{D}{\Longrightarrow}_i c_{j+1}$. A *global run* σ starting from a global configuration $g_0 = c_0^0c_0^1c_0^2...c_0^n$ is a (potentially infinite) set of local runs. Initially, σ contains local runs for n processes, with each starting from initial local configuration c_0^i for $0 \leq i \leq n$. Whenever a SM-DPDS responsible for a local run spawns a new process with the DCLIC $p_2\omega_2\theta_2$, a local run starting from $(\langle p_2, \omega_2\rangle, \theta_2)$ is added to σ.

2.3 From SM-DPN to DPN

We show in this section that every SM-DPN model is equivalent to a non self-modifying DPN as defined in [25]. Since the number of phases is finite, we can show that encoding every phase into the state set gives an equivalent DPN. Our translation follows the logic of the translation given in [20].

Let $\mathcal{M} = (\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$ be a SM-DPN for $n \in \mathbb{N}$, such that for $i \leq n$, $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i, \Delta_i^c)$ is a SM-DPDS. We can construct an equivalent DPN $\mathcal{M}' = (\mathcal{P}'_1, \mathcal{P}'_2, \ldots, \mathcal{P}'_n)$, where for $i \leq n$, $\mathcal{P}'_i = (P'_i, \Gamma_i, \Delta'_i)$ is a DPDS equivalent to $\mathcal{P}_i$, such that $P'_i = P_i \times 2^{\Delta_i \cup \Delta_i^c}$ and Δ'_i is computed as follows. Initially, Δ'_i is empty. For every $r \in \Delta_i \cup \Delta_i^c$ and for every phase $\theta \in 2^{\Delta_i \cup \Delta_i^c}$, such that $r \in \theta$:

1. if $r = p\gamma \hookrightarrow p_1\omega_1$, then add $(p, \theta)\gamma \hookrightarrow (p_1, \theta)\omega_1 \in \Delta'_i$;
2. if $r = p\gamma \hookrightarrow p_1\omega_1 \triangleright p_2\omega_2\theta_2$, then add $(p, \theta)\gamma \hookrightarrow (p_1, \theta)\omega_1 \triangleright (p_2, \theta_2)\omega_2 \in \Delta'_i$;
3. if $r = p \xhookrightarrow{(\rho_1, \rho_2)} p_1$ and $\rho_1 \subseteq \theta$, then for every $\gamma \in \Gamma$, add $(p, \theta)\gamma \hookrightarrow (p_1, (\theta \backslash \rho_1) \cup \rho_2) \in \Delta'_i$.

This algorithm terminates because we have a finite number of rules and hence, a finite number of phases. We can show that:

Proposition 1. *Let $(\langle p, \omega\rangle, \theta)$ and $(\langle p_1, \omega_1\rangle, \theta_1)$ be configurations of $\mathcal{P}_i$, and $D \subseteq \mathcal{D}_i$. $(\langle p, \omega\rangle, \theta) \overset{D}{\Longrightarrow}_{\mathcal{M}} (\langle p_1, \omega_1\rangle, \theta_1)$ iff $\langle (p, \theta), \omega\rangle \overset{D'}{\Longrightarrow}_{\mathcal{M}'} \langle (p_1, \theta_1), \omega_1\rangle$, such that $D' = \{(p_2, \theta_2)\omega_2 \mid p_2\omega_2\theta_2 \in D\}$*

Thus, we get:

Theorem 1. *Let $\mathcal{M} = (\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$ be a SM-DPN for $n \in \mathbb{N}$, such that for $i \leq n$, $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i, \Delta_i^c)$ is a SM-DPDS. We can construct an equivalent DPN $\mathcal{M}' = (\mathcal{P}'_1, \mathcal{P}'_2, \ldots, \mathcal{P}'_n)$, where for $i \leq n$, $\mathcal{P}'_i = (P'_i, \Gamma_i, \Delta'_i)$ such that $|P'_i| = |P_i| \cdot 2^{O(|\Delta_i| + |\Delta_i^c|)}$ and $|\Delta'_i| = (|\Delta_i| + |\Delta_i^c||\Gamma|) \cdot 2^{O(|\Delta_i| + |\Delta_i^c|)}$*

2.4 Modelling Self-Modifying Concurrent Code with SM-DPN

We give in this section a general process of converting a binary executable containing self-modifying code and concurrency into a SM-DPN. We suppose that we have an oracle that translates a binary program into a Control Flow Graph (CFG), such that each CFG transition corresponds to one instruction. One can obtain such an oracle using existing tools like Jakstab [19], IDA Pro [15], Radare2 [2], or ANGR [23].

We use the translation of [24] that models non self-modifying *sequential* instructions of the program by a standard PDS. We refer the reader to [24] for more details on this translation.

A self-modifying CFG transition that writes a binary value `v` to an address `d`, where `d` is the destination address for an executable region and `v` is a new value. Let ρ_1 be the set of rules obtained from translating instructions at `d` before self-modification and let ρ_2 be the set of rules obtained at `d` after the memory is modified. Suppose the CFG transition starts at control location p and leads to a control location p'. In this case, we add a rule $p \xhookrightarrow{(\rho_1, \rho_2)} p'$ to the SM-DPDS model.

If a CFG transition spawns a new thread, we add a rule $p\gamma \hookrightarrow p'\omega \triangleright p_2\omega_2\theta_2$, where $(\langle p_2, \omega_2\rangle, \theta)$ is the initial configuration of the newly created process.

2.5 LTL and Büchi Automata

In this section, we consider standard LTL formulas and Büchi Automata. Let AP be a set of atomic propositions.

Definition 2. *An LTL formula ψ is defined as follows (where $a \in AP$):*

$$\psi ::= \top \mid \bot \mid a \mid \neg\psi \mid \psi \wedge \psi \mid \boldsymbol{X}\psi \mid \psi \boldsymbol{U} \psi$$

Let $w = \alpha_0\alpha_1\alpha_2\ldots$ be an ω-word over 2^{AP} and ψ be an LTL formula. Let $w^i = \alpha_i\alpha_{i+1}\ldots$ be the subsequence of w starting from the i-th symbol, where $i \geq 0$. The satisfiability $w \models \psi$ is defined as follows: $w \models \top$; $w \not\models \bot$; $w \models a$ iff $a \in \alpha_0$; $w \models \neg\psi$ iff $w \not\models \psi$; $w \models \psi_1 \wedge \psi_2$ iff $w \models \psi_1$ and $w \models \psi_2$; $w \models \mathbf{X}\psi$ iff $w^1 \models \psi$; $w \models \psi_1 \mathbf{U} \psi_2$ iff there exists $k \geq 0$ such that for $j < k$, $w^j \models \psi_1$, and $w^k \models \psi_2$. We define the *eventually* operator as follows: $\mathbf{F}\psi = \top\mathbf{U}\psi$, which means that ψ will hold at some point of the run. The *globally* operator $\mathbf{G}\psi = \neg\mathbf{F}\neg\psi$ means that ψ holds universally along the run.

Definition 3. *A Büchi Automaton is a tuple $\mathcal{B} = (G, \Sigma, T, g_0, F)$, where G is a finite set of states, $T \subseteq G \times \Sigma \times G$ is the set of transitions, $g_0 \in G$ is the initial state, and $F \subseteq G$ is the set of accepting states.*

For an ω-word $w = \alpha_0\alpha_1\alpha_2\ldots$, such that $\alpha_i \in \Sigma, i \geq 0$, a *run* on a BA $\mathcal{B}$ is an infinite sequence $r = g_0g_1g_2\ldots$, such that $(g_{i-1}, \alpha_{i-1}, g_i) \in T$ for $i \geq 1$. A run r is *accepting* if the run visits some accepting state $g_f \in F$ infinitely often. $\mathcal{B}$ accepts an infinite word iff there is an accepting run on $\mathcal{B}$. It is well known that given an LTL formula ψ, we can construct a Büchi Automaton (BA) $\mathcal{B}_\psi$ on words over $\Sigma = 2^{AP}$ that accepts all ω-words that satisfy ψ [28].

2.6 Single-Indexed LTL for SM-DPNs

Definition 4. *A Single-indexed LTL formula is a formula of the form $f = \bigwedge_{i=1}^{n} f_i$, where f_i is a standard LTL formula over AP_i.*

Let us consider a SM-DPN $\mathcal{M} = (\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$, such that $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i, \Delta_i^c)$, $1 \leq i \leq n$, and a single-indexed LTL formula $f = \bigwedge_{i=1}^{n} f_i$ over $AP = \bigcup_{i=1}^{n} AP_i$, where f_i is an LTL formula over AP_i for $\mathcal{P}_i$ and a labelling function $\lambda_i : P_i \to 2^{AP_i}$. For each control location $p \in \bigcup_{i=1}^{n} P_i$, let $\pi(p)$ be a function that maps p to the index i of its corresponding SM-DPDS, or $\pi(p) = i$ if $p \in P_i$. For a local run $\sigma = (\langle p_0, \omega_0\rangle, \theta_0)(\langle p_1, \omega_1\rangle, \theta_1)\ldots$, let $\pi(\sigma) = \pi(p_0)$. Let $\tau = \sigma_0\sigma_1\ldots$ be a global run on $\mathcal{M}$, where for $j \geq 0$, σ_j is a local run in τ. We define the satisfiability condition for a global run $\tau = \sigma_0\sigma_1\ldots$ on SM-DPN $\mathcal{M} = (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n)$ and a single-indexed LTL formula $f = \bigwedge_{i=0}^{n} f_i$:

Definition 5. *$\tau \models f$ if for every local run $\sigma_j \in \tau$, where $j \geq 0$:*

1. *there exists $D_j = \{p_j^0\omega_j^0\theta_j^0, p_j^1\omega_j^1\theta_j^1, \ldots, p_j^m\omega_j^m\theta_j^m\}$, such that $\sigma_j \models_{D_j} f_{\pi(\sigma_j)}$,*
2. *for every $0 \leq k \leq m$ there exists a local run σ_j^k starting from $(\langle p_j^k, \omega_j^k\rangle, \theta_j^k)$, such that $\tau \cup \{\sigma_j^k\} \models f$.*

For $\sigma = (\langle p_0, \omega_0\rangle, \theta_0)(\langle p_1, \omega_1\rangle, \theta_1)\ldots$, $\sigma \models_D f_i$ if the ω-word $w = \lambda_i(p_0)\lambda_i(p_1)\ldots$ satisfies f_i and σ spawns new processes with DCLICs D. For a local configuration c, $c \models_D f_i$ if there is a local run $\sigma_c = cc_2c_3\ldots$, such that $\sigma_c \models_D f_i$.

3 LTL Model-Checking for SM-DPNs

From now on, we fix a SM-DPN $\mathcal{M} = (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n)$ be a SM-DPN, where for $0 \leq i \leq n$, $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i, \Delta_i^c)$ is a SM-DPDS, and a single-indexed LTL formula $f = \bigwedge_{i=0}^{n} f_i$, such that f_i is an LTL formula corresponding to $\mathcal{P}_i$, and let AP_i be the set of all atomic propositions used in f_i. Let $\mathcal{B}_i = (G_i, 2^{AP_i}, T_i, g_i^0, F_i)$ be a Büchi automaton for the corresponding LTL formula f_i. Model checking a global configuration $\mathcal{G}$ of $\mathcal{M}$ over f is not trivial for two reasons. First, it is not enough to check every local run starting from local configurations in $\mathcal{G}$, because we also need to check spawned processes. Second, model checking every possibly spawned process is too restrictive, because not all processes are required to be spawned during an accepting run, so even if they violate their LTL formulas, the global run would still be accepting if it does not spawn such processes. Therefore, when model checking $\mathcal{P}_i$ over f_i, it is important to remember which processes were spawned during the accepting run. We divide the problem of checking whether a global run starting from $\mathcal{G}$ satisfies f into the following steps:

1. We compute a Self-Modifying Büchi Dynamic Pushdown System (SM-BDPDS) $\mathcal{BP}_i$ for each pair of $\mathcal{P}_i$ and $\mathcal{B}_i$, $1 \leq i \leq n$.
2. For each $\mathcal{BP}_i$, we compute sets of pairs of local configurations and DCLICs of form $\big(c, D\big)$, such that $c \models_D f_i$.
3. To be able to compute such sets, we need to be able to finitely represent such pairs. For each $\mathcal{BP}_i$ we will construct a MA $\mathcal{A}_i$, such that it accepts all configurations c of $\mathcal{P}_i$ satisfying f_i and produces a set of DCLICs D spawned during an accepting run, or $c \models_D f_i$.
4. Using the MAs $\mathcal{A}_i$, we compute the maximal set of DCLICs $\mathcal{D}_{fp}$, such that it contains only DCLICs for local configurations that satisfy their LTL formulas, and that the accepting runs on that configurations spawn only DCLICs in $\mathcal{D}_{fp}$.
5. For every local configuration $(\langle p, \omega\rangle, \theta) \in \mathcal{G}$, we use the MA $\mathcal{A}_{\pi(p)}$ to check that there exists $D' \subseteq \mathcal{D}_{fp}$, such that $(\langle p, \omega\rangle, \theta) \models_{D'} f_{\pi(p)}$.

The first and second steps reduce the LTL satisfiability problem of a single process into the emptiness problem of SM-BDPDS. Then, the third step reduces the emptiness problem of SM-BDPDS to the reachability analysis of SM-BDPDS using Multi Automata (MA). The fourth step filters out DCLICs that can not satisfy f. And the fifth step uses filtered DCLICs to check whether a global configuration can satisfy f without spawning unsatisfiable DCLICs.

4 LTL Model-Checking for SM-DPDS

This section describes how to solve the LTL satisfiability problem of a single process by reducing it to the reachability analysis of Self-Modifying Büchi Dynamic Pushdown Systems (SM-BDPDS). As can be seen in Section 3, we do not only need to check that a SM-DPDS has a valid run, but also know what processes are spawned during the run.

4.1 Self-Modifying Büchi Dynamic Pushdown Systems

It is common for an automata-based LTL model checking to compute a product of a standard PDS and a Büchi Automaton, which is often called Büchi Pushdown System (BPDS) [7, 13]. Similarly, Self-Modifying Büchi Dynamic Pushdown Systems (SM-BDPDS) is a product of a SM-DPDS and a BA:

Definition 6. *A Self-Modifying Büchi Dynamic Pushdown Systems (SM-BDPDS) $\mathcal{BP} = (P, \Gamma, \Delta, \Delta^c, I, F)$ is a SM-DPDS with extra elements $I \subseteq P$, which is a set of initial states, and $F \subseteq P$, which is a set of accepting states.*

For a SM-BDPDS $\mathcal{BP} = (P, \Gamma, \Delta, \Delta^c, I, F)$, a local configuration $c_0 = (\langle p_0, \omega_0 \rangle, \theta_0)$ and a set of DCLICs D, (c_0, D) satisfies $\mathcal{BP}$ if $p_0 \in I$ and there is a local run $\sigma = c_0 c_1 c_2 \dots$ in $\mathcal{BP}$, such that $\mathcal{BP}$ spawns new processes with DCLICs D during σ and there is an infinite subsequence of configurations $c_{k_0} c_{k_1} \dots$, where $c_{k_j} = (\langle p_{k_j}, \omega_{k_j} \rangle, \theta_{k_j})$ for $j \geq 0$ and $p_{k_j} \in F$. We denote the set of all configurations and DCLICs accepted by $\mathcal{BP}$ as $\mathcal{L}(\mathcal{BP})$.

For a SM-DPDS $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i, \Delta_i^c)$, a Büchi Automaton $\mathcal{B}_i = (G, 2^{AP}, T, g_0, F)$ that corresponds to an LTL formula f_i, and a labelling function λ_i, we can compute a SM-BDPDS $\mathcal{BP}_i = (P_i', \Gamma_i, \Delta_i', \Delta_i^{c'}, I_i', F_i')$, where $P_i' = P_i \times G$, $I_i' = P_i \times \{g_0\}$, and $F_i' = P_i \times F$. Let $p, p_1, \in P_i$, $g_1, g_2 \in G_i$, $\gamma \in \Gamma$, $\omega_1 \in \Gamma^*$, $\rho_1, \rho_2 \subseteq \Delta_i \cup \Delta_i^c$, $p_2 \in P_j$, $\omega_2 \in \Gamma_j^*$, $\theta_2 \in 2^{\Delta_j \cup \Delta_j^c}$. Initially, $\Delta_i' = \Delta_i^{c'} = \emptyset$. We construct rules for $\mathcal{BP}_i$ as follows:

1. $[p, g_1]\gamma \hookrightarrow [p_1, g_2]\omega_1 \in \Delta_i'$ iff $p\gamma \hookrightarrow p_1\omega_1 \in \Delta_i$ and $(g_1, \lambda_i(p), g_2) \in T_i$,
2. $[p, g_1]\gamma \hookrightarrow [p_1, g_2]\omega_1 \triangleright p_2\omega_2\theta_2 \in \Delta_i'$ iff $p\gamma \hookrightarrow p_1\omega_1 \triangleright p_2\omega_2\theta_2 \in \Delta_i$ and $(g_1, \lambda_i(p), g_2) \in T_i$,
3. $[p, g_1] \xrightarrow{(\sigma_1, \sigma_2)} [p_1, g_2] \in \Delta_i^{c'}$ iff $p \xrightarrow{(\rho_1, \rho_2)} p_1 \in \Delta_i^c$, $(g_1, \lambda_i(p), g_2) \in T_i$, $\sigma_1 = prod(\rho_1)$, and $\sigma_2 = prod(\rho_2)$, where $prod(\rho)$ is a set of rules obtained from $\rho \subseteq \Delta_i \cup \Delta_i^c$.

Intuitively, $\mathcal{BP}_i$ is a product of $\mathcal{P}_i$ and the BA $\mathcal{B}_i$. The behavior of the constructed SM-BDPDS $\mathcal{BP}_i$ is the same as of $\mathcal{P}_i$ synchronized with $\mathcal{B}_i$ for the LTL formula f_i. The intuition behind this construction is that if there is a run on SM-BDPDS $\sigma = (\langle [p_0, g_0], \omega_0 \rangle, \theta_0)(\langle [p_1, g_1], \omega_1 \rangle, \theta_1) \dots$, then there should be a valid run $\sigma^p = (\langle p_0, \omega_0 \rangle, \theta_0)(\langle p_0, \omega_0 \rangle, \theta_0) \dots$ spawning DCLICs D on $\mathcal{P}_i$ and a valid run $\sigma^b = g_0 g_1 \dots$ on $\mathcal{B}_i$. Therefore, $\sigma^p \models_D f_i$ iff $\big((\langle [p_0, g_0], \omega_0 \rangle, \theta_0), D\big) \in \mathcal{L}(\mathcal{BP}_i)$.

4.2 LTL Satisfiability to the Emptiness Problem of SM-DPDS

To compute satisfiability of a single process we extend the automata-based approach for non self-modifying DPNs proposed by [25].

Theorem 2. *For a SM-BDPDS $\mathcal{BP}_i = (P_i', \Gamma_i, \Delta_i', \Delta_i^{c'}, I_i', F_i')$, $\big((\langle p, \omega \rangle, \theta), D\big) \in \mathcal{L}(\mathcal{BP}_i)$ iff $\exists D_1, D_2, D_3 \subseteq \mathcal{D}_i$, s.t. $D = D_1 \cup D_2 \cup D_3$ and the following conditions hold:*

α_1 $(\langle p, \omega \rangle, \theta) \xrightarrow{D_1}{}^* (\langle p', \gamma\omega' \rangle, \theta')$ *for some* $p' \in P_i'$, $\theta, \theta' \subseteq (\Delta_i' \cup \Delta_i^{c'})$, $\gamma \in \Gamma_i$, $\omega' \in \Gamma^*$, *and*

α_2 $(\langle p', \gamma\rangle, \theta') \xRightarrow{D_2}{}^{+} (\langle g, u\rangle, \theta'')$ *and* $(\langle g, u\rangle, \theta'') \xRightarrow{D_3}{}^{*} (\langle p', \gamma v\rangle, \theta')$ *for some* $g \in F_i'$, $\theta'' \subseteq (\Delta_i' \cup \Delta_i^{c'})$, $u, v \in \Gamma^*$.

Let us explain the intuition behind the theorem. Since $\big((\langle p, \omega\rangle, \theta), D\big) \in \mathcal{L}(\mathcal{BP}_i)$, there must be an infinite sequence of local configurations with an accepting state g. This sequence is produced by an infinite cycle starting from some $(\langle p', \gamma\rangle, \theta')$, visiting $(\langle g, u\rangle, \theta'')$, and then going to $(\langle p', \gamma v\rangle, \theta')$. Since rules of $\mathcal{BP}_i$ only look at the top symbol of the stack content, $\mathcal{BP}_i$ can apply the same rules on $(\langle p', \gamma v\rangle, \theta')$ and end up at $(\langle p', \gamma vv\rangle, \theta')$ and so on. During this cycle, $\mathcal{BP}_i$ spawns processes with DCLICs $D_2 \cup D_3$ (D_2 to reach $(\langle g, u\rangle, \theta'')$ and D_3 to go back). This is ensured by the condition α_1. Moreover, the starting state $(\langle p, \omega\rangle, \theta)$ must be backwards reachable from $(\langle p', \gamma\rangle, \theta')$, which is ensured by the condition α_1. Assume that $\mathcal{BP}_i$ spawns processes with DCLICs D_1 along the path from $(\langle p, \omega\rangle, \theta)$ to $(\langle p', \gamma\rangle, \theta')$. Therfore, $\mathcal{BP}_i$ spawns $D_1 \cup D_2 \cup D_3$ during the accepting run and hence, $\big((\langle p, \omega\rangle, \theta), D\big)$ is accepted by $\mathcal{BP}_i$.

Corollary 1. *For a SM-BDPDS* $\mathcal{BP}_i = (P_i', \Gamma_i, \Delta_i', \Delta_i^{c'}, F_i)$, $\big((\langle p\omega\rangle, \theta), D\big) \in \mathcal{L}(\mathcal{BP}_i)$ *iff* $\exists D_1, D_2' \subseteq \mathcal{D}_i$, $\exists \theta' \subseteq \Delta_i' \cup \Delta_i^{c'}$, *s.t.* $D = D_1 \cup D_2'$ *and the following conditions hold:*

β_1 $\big((\langle p, \omega\rangle, \theta), D\big) \in pre^*(\{p'\} \times \gamma\Gamma_i^* \times \{\theta'\} \times \{\emptyset\})$, *and*

β_2 $\big((\langle p', \gamma\rangle, \theta'), D_2'\big) \in pre^+((F_i \times \Gamma_i^* \times 2^{\Delta_i' \cup \Delta_i^{c'}} \times 2^{\mathcal{D}_i}) \cap pre^*(\{p'\} \times \gamma\Gamma_i^* \times \{\theta'\} \times \{\emptyset\}))$.

The corollary is a rewording of Theorem 2 using *pre* notation instead of successor relationship, and where D_2' equals to $D_2 \cup D_3$. Intuitively, for the condition β_1, if $(\langle p, \omega\rangle, \theta) \xRightarrow{D_1}{}^{*} (\langle p', \gamma\omega'\rangle, \theta')$, then $\big((\langle p, \omega\rangle, \theta), D\big) \in pre^*(\{p'\} \times \gamma\Gamma_i^* \times \{\theta'\} \times \{\emptyset\})$. And for condition β_2, if $(\langle g, u\rangle, \theta'') \xRightarrow{D_3}{}^{*} (\langle p', \gamma v\rangle, \theta')$, then $\big((\langle g, u\rangle, \theta''), D_3\big) \in (F_i \times \Gamma_i^* \times 2^{\Delta_i' \cup \Delta_i^{c'}} \times 2^{\mathcal{D}_i}) \cap pre^*(\{p'\} \times \gamma\Gamma_i^* \times \{\theta'\} \times \{\emptyset\})$. And since $(\langle p', \gamma\rangle, \theta') \xRightarrow{D_2}{}^{+} (\langle g, u\rangle, \theta'')$, then $\big((\langle p', \gamma\rangle, \theta'), D_2'\big)$ is in pre^+ of all such $\big((\langle g, u\rangle, \theta''), D_3\big)$, where $D_2' = D_2 \cup D_3$.

5 Automata-Based Model Checking

As shown in Section 4, the LTL model checking of SM-DPNs boils down to the efficient computation of the set of predecessors pre^* for local configurations and sets of DCLICs of a SM-DPDS. We follow a similar approach as [20, 25] and give in this section the efficient algorithm for computing this set using Multi Automata.

5.1 Multi Automata

Multi Automata (MA) are widely used to finitely represent potentially infinite sets of local configurations and DCLICs. Following [25], we use MA to represent sets of local configurations and DCLICs of SM-DPDS:

Definition 7. *A Multi-Automaton is a tuple* $\mathcal{A}_i = (Q_i, \Gamma_i, \delta_i, I_i, Acc_i)$*, where* Q_i *is a finite set of states,* Γ_i *is the alphabet of* $\mathcal{P}_i$*,* $\delta_i \subseteq Q_i \times (\Gamma_i \cup \{\varepsilon\}) \times 2^{\mathcal{D}_i} \times Q_i$ *is a set of transitions,* $I_i = P_i \times 2^{\Delta_i \cup \Delta_i^c} \subseteq Q_i$ *is a set of initial states representing the control point and the phase of the configuration, and* $Acc_i \subseteq Q_i$ *is a set of accepting states. Transition* $(q, \gamma, D, q') \in \delta_i$ *can be denoted as* $q \xrightarrow{\gamma/D}_i q'$*.*

The reflexive-transitive closure $q \xrightarrow{\omega/D}_i{}^* q'$ for transitions is defined as: (1) $q \xrightarrow{\varepsilon/\emptyset}_i{}^* q$, (2) if $q \xrightarrow{\gamma/D}_i q''$ and $q'' \xrightarrow{\omega/D'}_i{}^* q'$, then $q \xrightarrow{\gamma\omega/D\cup D'}_i{}^* q'$, where $\omega \in \Gamma_i^*$, $D' \subseteq \mathcal{D}_i$.

For a local configuration $(\langle p, \omega\rangle, \theta)$ and a set of DCLICs D, the MA accepts tuples $\big((\langle p, \omega\rangle, \theta), D\big)$ iff there is a path $(p, \theta) \xrightarrow{\omega/D}_i{}^* q_f$ for some $q_f \in Acc_i$. From now on, we will omit the index i for $\rightarrow_i$ when it is understood from the context.

Let $\mathcal{L}(\mathcal{A}_i)$ be the set of configurations and DCLICs accepted by the MA $\mathcal{A}_i$. We say that a set of configurations and DCLICs W is *regular* iff there exists a MA $\mathcal{A}_i$, such that $W = \mathcal{L}(\mathcal{A}_i)$.

5.2 Algorithm for pre^* Computation

We prove in this section that given a SM-DPDS $\mathcal{P}_i$ and a regular set of pairs of configurations and DCLICs W accepted by a MA $\mathcal{A}_i$, we can compute a new MA $\mathcal{A}_i^{pre^*}$ that accepts $pre^*(W)$. We introduce a saturation algorithm that extends original MA into the new MA that also accepts $pre^*(W)$. Let $\mathcal{A}_i = (Q_i, \Gamma_i, \delta_i, I_i, Acc_i)$ be the original MA that accepts configurations of a SM-DPDS $\mathcal{P}_i = (P_i, \Gamma_i, \Delta_i, \Delta_i^c)$.

Without loss of generality, we can assume that there are no rules that remove themselves. In other words, there are no rules of type $r = p \xhookrightarrow{(\rho_1,\rho_2)} p' \in \Delta_i^c$, such that $r \in \rho_1$. It is possible because we can substitute such rules with two new rules: $r = p \xhookrightarrow{(\emptyset,\emptyset)} p^r$ and $r' = p^r \xhookrightarrow{(\rho_1,\rho_2)} p'$, where r' is a new rule and p^r is a new state. We need this restriction because the algorithm assumes that a configuration $(\langle p, \omega\rangle, \theta)$ can be reached from every direct predecessor by applying some rule in θ.

Now, we can construct $\mathcal{A}_i^{pre^*} = (Q_i, \Gamma_i, \delta_i', I_i, Acc_i)$. Initially, $\delta_i' = \delta_i$. Then, we apply the following saturation rules:

- μ_1 If $r = p\gamma \hookrightarrow p'\omega \in \Delta_i$, then for every $\theta \in 2^{\Delta_i \cup \Delta_i^c}$, s.t. $r \in \theta$ and $(p', \theta) \xrightarrow{\omega/D}{}^* q \in \delta_i'$, add transition $(p, \theta) \xrightarrow{\gamma/D} q$ to δ_i';
- μ_2 if $r = p\gamma \hookrightarrow p'\omega \triangleright p''\omega''\theta'' \in \Delta_i$, then for every $\theta \in 2^{\Delta_i \cup \Delta_i^c}$, s.t. $r \in \theta$ and $(p', \theta) \xrightarrow{\omega/D}{}^* q \in \delta_i'$, add transition $(p, \theta) \xrightarrow{\gamma/D\cup\{p''\omega''\theta''\}} q$ to δ_i';
- μ_3 If $r = p \xhookrightarrow{(\rho_1,\rho_2)} p' \in \Delta_i^c$, then for every $\theta' \in 2^{\Delta_i \cup \Delta_i^c}$ and $\gamma \in \Gamma_i$, s.t. $r \in \theta'$, $\rho_2 \subseteq \theta'$ and $(p', \theta') \xrightarrow{\gamma/D} q$, then add $(p, \theta) \xrightarrow{\gamma/D} q$, such that $\theta' = (\theta \backslash \rho_1) \cup \rho_2$.

This procedure terminates because there is a finite number of transitions we can add, which is $(|P_i||2^{\Delta_i \cup \Delta_i^c}| + Q_i)^2 |\Gamma \cup \{\varepsilon\}||\mathcal{D}_i|$.

Let us explain the intuition behind the saturation rules. Rule μ_1 adds predecessors obtained from standard pushdown rules. Consider a rule $r = p\gamma \hookrightarrow p'\omega \in \Delta_i$ and let us consider a path of $\mathcal{A}_i^{pre^*}$ of the form $(p', \theta) \xrightarrow{\omega/D}{}^* q \xrightarrow{\omega'/D'}{}^* q_f$ such that $q_f \in Acc_i$ and $r \in \theta$, which means $(\langle p', \omega\omega'\rangle, \theta, D \cup D') \in \mathcal{L}(\mathcal{A}_i^{pre^*})$. Since the phase θ contains r, then $(\langle p, \gamma\omega'\rangle, \theta)$ is a direct predecessor of $(\langle p', \omega\omega'\rangle, \theta)$. Thus, since r does not spawn new processes, there is no need tp update the DCLICs D. Therefore, we add the new transition $(p, \theta) \xrightarrow{\gamma/D} q$ to $\mathcal{A}_i^{pre^*}$, so that $(\langle p, \gamma\omega'\rangle, \theta, D \cup D')$ will be accepted by the path $(p, \theta) \xrightarrow{\gamma/D} q \xrightarrow{\omega'/D'}{}^* q_f$ in $\mathcal{A}_i^{pre^*}$.

Similarly, rule μ_2 adds predecessors that require a process to be spawned. In this case, consider a rule $r = p\gamma \hookrightarrow p'\omega \triangleright p''\omega''\theta'' \in \Delta_i$ and let there be a path of the form $(p', \theta) \xrightarrow{\omega/D}{}^* q \xrightarrow{\omega'/D'}{}^* q_f$ such that $q_f \in Acc_i$ and $r \in \theta$. Therefore, $(\langle p', \omega\omega'\rangle, \theta, D \cup D') \in \mathcal{L}(\mathcal{A}_i^{pre^*})$. Since the phase θ contains r, then $(\langle p, \gamma\omega'\rangle, \theta)$ is a direct predecessor of $(\langle p', \omega\omega'\rangle, \theta)$. Since r spawns a new process with the DCLIC $p''\omega''\theta''$, we add $p''\omega''\theta''$ to the DCLIC D. Therefore, we add the new transition $(p, \theta) \xrightarrow{\gamma/D \cup \{p''\omega''\theta''\}} q$, so that $(\langle p, \gamma\omega''\rangle, \theta, D \cup D' \cup \{p''\omega''\theta''\})$ will be accepted by the path $(p, \theta) \xrightarrow{\gamma/D \cup \{p''\omega''\theta''\}} q \xrightarrow{\omega''/D'}{}^* q_f$.

The saturation rule μ_3 adds the predecessors that are obtained by self-modifying rules. Consider a rule $r = p \xhookrightarrow{(\rho_1, \rho_2)} p' \in \Delta_i^c$ and a path in $\mathcal{A}_i^{pre^*}$ of the form $(p', \theta') \xrightarrow{\gamma/D} q \xrightarrow{\omega''/D'}{}^* q_f$, such that $q_f \in Acc_i$, $r \in \theta'$, $\rho_2 \subseteq \theta'$. Therefore, $(\langle p', \gamma\omega\rangle, \theta', D \cup D') \in \mathcal{L}(\mathcal{A}_i^{pre^*})$. Since the phase θ' contains r and all rules in ρ_2, then $(\langle p, \gamma\omega\rangle, \theta)$ is a direct predecessor of $(\langle p', \gamma\omega\rangle, \theta')$, where $\theta = (\theta' \backslash \rho_2) \cup \rho_1$. Since this transition does not spawn any new processes, there is no need to update DCLICs D. Therefore, we add the new transition $(p, \theta) \xrightarrow{\gamma/D} q$ to $\mathcal{A}_i^{pre^*}$, so that $(\langle p, \gamma\omega''\rangle, \theta)$ will be accepted by the path $(p, \theta) \xrightarrow{\gamma/D} q \xrightarrow{\omega''/D'}{}^* q_f$ in $\mathcal{A}_i^{pre^*}$.

Lemma 1. *Given a regular set of configurations and DCLICs W recognized by MA $\mathcal{A}_i = (Q_i, \Gamma_i, \delta_i, I_i, Acc_i)$, we can effectively compute MA $\mathcal{A}_i^{pre^*} = (Q_i, \Gamma_i, \delta_i', I_i, Acc_i)$, such that $\mathcal{L}(\mathcal{A}_i^{pre^*}) = pre^*(W)$.*

We can show that the saturation algorithm described above computes MA $\mathcal{A}_i^{pre^*}$ that accepts pre^* of a regular set of configurations and DCLICs (the proof can be found in Appendix A). We use Lemma 1 to show:

Theorem 3. *For a regular set of local configurations and DCLICs $W \subseteq P_i \times \Gamma_i^* \times 2^{\Delta_i \cup \Delta_i^c} \times 2^{\mathcal{D}_i}$, the set $pre^*(W)$ is also regular and can be effectively computed.*

6 Effective Algorithm for Model Checking SM-DPNs

Now, we are ready to tackle the main problem of this paper - effective LTL model checking of SM-DPNs. We combine the concepts presented in Sections 3, 4, and 5.2 to give a full algorithm for LTL model checking SM-DPNs.

Consider a SM-DPN $\mathcal{M} = (\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$ and a single-indexed LTL formula $f = \bigwedge_{i=0}^{n} f_i$. From Theorem 3 and Corollary 1, given a SM-DPDS $\mathcal{P}_i$ and an LTL formula f_i we can construct a MA $\mathcal{A}_i$ to determine whether or not $\mathcal{P}_i$ satisfies f_i. To construct $\mathcal{A}_i$, we iterate over possible p',γ', and θ'. We construct the MA for $pre^*(\{p'\} \times \{\theta'\} \times \gamma' \Gamma_i^* \times \{\emptyset\})$. $\{p'\} \times \{\theta'\} \times \gamma' \Gamma_i^* \times \{\emptyset\}$ is a regular set because $\gamma' \Gamma_i^*$ is a regular word and all other items are finite sets. Then, we compute the intersection with $(F_i \times 2^{\Delta_i' \cup \Delta_i^{c'}} \times \Gamma_i^* \times 2^{\mathcal{D}_i})$. This set is also regular because Γ_i^* is a regular word and other sets are finite. After that, we contruct pre^+ MA for the intersection and test $(\langle p', \gamma' \rangle \theta', D_2')$ on the pre^+. If the pre^+ automaton accepts $(\langle p', \gamma' \rangle \theta', D_2')$, then we add the transition $(p', \theta') \xrightarrow{\gamma'/D_2'} q_f$ to the MA $\mathcal{A}_i'$, where q_f is a final state. Next, we compute pre^* on the $\mathcal{A}_i'$ to get the final automaton $\mathcal{A}_i = pre^*(\mathcal{A}_i')$. From Corollary 1, this $\mathcal{A}_i$ is such that if $(c, D) \in \mathcal{L}(\mathcal{A}_i)$, then $c \models_D f_i$.

Then, we use Theorem 4 for find $\mathcal{D}_{fp}$. This theorem requires that given a configuration $(\langle p, \omega \rangle, \theta)$ and a set of DCLICs D, we need to know whether $(\langle p, \omega \rangle, \theta) \models_D f_\pi(c)$. We can use the MA $\mathcal{A}_i$ constructed by the method above for that purpose. And after computing $\mathcal{D}_{fp}$, we proceed with the algorithm in Section 3 to get whether a global configuration satisfies f.

Consider a SM-DPN $\mathcal{M} = (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n)$ and First, we need to compute the maximal set $\mathcal{D}_{fp}$. Let $F : 2^{\bigcup_{k=1}^{n} D_k} \to 2^{\bigcup_{k=1}^{n} D_k}$ be defined as $F(D) = \{p\omega\theta \in D_I \mid \exists D' \subseteq D : (\langle [p, g^0_{\pi(p)}], \omega \rangle, \theta) \models_D f_{\pi(i)}\}$. Let $D^0 D^1 D^2 \ldots$ be a sequence generated by the recursive application of F, such that $D^0 = D_I$, and $D^j = F(D^{j-1})$ for $j \geq 1$. We can show that $\mathcal{D}_{fp} = \bigcap_j D^j$ and can be effectively computed by F:

Theorem 4. *We can effectively compute $\mathcal{D}_{fp}$, s.t. for every DCLIC $p\omega\theta \in \bigcup_{k=1}^{n} \mathcal{D}_k$, $(\langle p, \omega \rangle, \theta) \models_{D'} f_{\pi(p)}$ iff $p\omega\theta \in \mathcal{D}_{fp}$ and $D' \subseteq \mathcal{D}_{fp}$.*

Intuitively, the function F takes a set of DCLICs D that *hypothetically* satisfy their corresponding LTL formulas. The function returns a smaller set of DCLICs D', where for every DCLIC $p'\omega'\theta' \in D'$, there exists another set of DCLICs $D'' \subseteq D$, such that $(\langle p', \omega' \rangle, \theta') \models_{D''} f_{\pi(p')}$. Initially, D^0 is the set of all DCLICs D_I. At the first step, we exclude DCLICs that can not satisfy f regardless of what DCLICs they generate (because we assume that every DCLIC satisfies f). Then, every next iteration excludes such DCLICs that spawn unsatisfiable DCLICs during their accepting runs. At the end, the function should plateau at a constant set of DCLICs that satisfy f. In other words, we can compute the greatest fixpoint $\mathcal{D}_{fp}$ on the recursion starting from D_I because this function reduces a countable set of DCLICs for each step. Therefore, we can use Theorem 4 to find the set of valid DCLICs for a single-indexed LTL formula.

Now, for a global configuration $\mathcal{G} = c_0 c_1 c_2 ... c_m$ and a single indexed formula $f = \bigwedge_{i=1}^{n} f_i$, we can determine whether the global configuration satisfies f. We simply check for each local configuration $c_j = (\langle p_j, \omega_j \rangle, \theta_j)$, if there exists $D \subseteq \mathcal{D}_{fp}$, such that $c_j \models_D f_\pi(p_j)$. $\mathcal{G} \models f$ if all configurations $c_j \in \mathcal{G}$ satisfy their corresponding $f_{\pi(p_j)}$.

7 Experiments

7.1 Comparison with Model Checking DPNs

Since SM-DPNs are equivalent to normal DPNs, we compared our *direct* model checking approach with the approach that consists of translating SM-DPN to the equivalent DPN and applying the algorithm proposed by [25]. First, we implemented both algorithms using Python 3. Then, we generated a set of random SM-DPN s and single-indexed formulas. Then, for each pair of SM-DPN and a formula we compared the time and memory needed for computing whether the SM-DPN satisfies the formula. We summarized our results in Table 1. The column $|\mathcal{M}|$ specifies the number of distinct SM-DPDS in the SM-DPN model, $|f_i|$ is the number of transitions in a BA obtained from the i-th LTL formula, $|\Delta_i|$ is the number of non self-modifying rules in each SM-DPDS, $|\Delta_i^c|$ is the number of self-modifying rules in each SM-DPDS, **T** is the time took for the algorithm and **mem** is the amount of memory used during the computation. The memory usage was recorded usign the Python's built-in `tracemalloc` package. The fields with *timeout* specify that the implementation took more than 10 hours to execute, and OOM (out-of-memory) means that the algorithm was terminated because there was not enough memory resource for the computation. The experiments were conducted on a laptop with a CPU AMD Ryzen 7 8845HS and 10 GB of available memory (8 GB RAM and 2 GB swap pages).

From Table 1, we can see that our algorithm performs consistently better in terms of time and memory than translating SM-DPNs into standard DPNs and applying the LTL algorithm for standard DPNs from [25]. We highlighted some cases where our algorithm performed significantly better. For example, when a SM-DPN contains 3 processes, 129 standard rules and 2 self-modifying rules, our direct LTL model checking took 769.27 seconds ($\sim$13 minutes), while the model checking of an equivalent DPN took almost 2 days. On average, our algorithm is 468 times faster for 3 self-modifying rules, and 95 times faster for 2 self-modifying rules.

7.2 LTL Model Checking SM-DPNs for Malware Specification

We evaluated the applicability of our approach for malware detection. We have collected samples of existing malware from malware databases, such as Virus Share [10] and Malware Bazaar [1]. We have found one sample of Mirai malware and one sample of Gozi malware that are both *concurrent* and *self-modifying*. We also considered one self-modifying version of a concurrent generic backdoor.

We translated the malware samples into SM-DPNs using the method described in Section 2.4. We obtained the CFGs of malware using ANGR analyzer [23]. We made use of its symbolic execution to resolve system calls as control points, identify self-modifying instructions, and compute states of spawned processes. We use system calls as atomic propositions AP. Our labelling function λ is that if a there is a system call x at a control point p, then $\lambda(p) = \{x\}$ and otherwise, $\lambda(p) = \emptyset$.

			Our approach		**SM-DPN to DPN**	
$\lvert\mathcal{M}\rvert$	$\lvert f_i\rvert$	$\lvert\Delta_i\rvert + \lvert\Delta_i^c\rvert$	**T**, sec	**mem**, KiB	**T**, sec	**mem**, KiB
1	5	10 + 3	0.01	63.57	0.4	1 153.93
1	6	10 + 3	3.98	236.29	2 927.27	2 017.56
1	7	10 + 3	2.87	325.34	815.80	1 232.78
1	8	10 + 3	5.29	674.29	8 894.93	2 420.37
1	5	10 + 1	0.30	63.57	7.23	134.47
1	6	10 + 1	0.10	63.57	3.11	116.18
1	7	10 + 1	**2.12**	642.30	**389.87**	1 045.70
2	1	12 + 2	1.24	71.24	21.76	561.31
2	1	22 + 2	1.53	78.41	39.49	706.41
2	1	32 + 2	**10.04**	179.01	**1 473.41**	1 542.21
2	12	32 + 2	114.78	1 433.08	timeout	timeout
3	4	31 + 3	29.19	483.58	3 586.07	10 920.32
3	5	32 + 5	0.02	72.42	4.21	41 818.15
3	5	42 + 5	1401.31	3 661.34	timeout	timeout
2	7	32 + 4	253.80	1 347.30	timeout	timeout
3	8	50 + 4	607.18	1 717.22	timeout	timeout
2	9	34 + 5	904.25	2 924.89	timeout	timeout
2	13	40 + 3	0.02	72.50	1.09	5 212.05
4	1	22 + 2	**31.92**	476.02	**4 644.77**	2 492.78
3	14	37 + 2	32.93	585.57	1 824.50	8 695.83
3	41	43 + 2	275.18	5 206.40	timeout	timeout
4	1	52 + 2	47.35	919.43	timeout	timeout
4	1	62 + 2	60.15	920.29	timeout	timeout
5	1	12 + 2	5.03	167.78	232.47	1 341.30
5	1	22 + 2	13.99	291.78	1 804.86	2 088.73
5	2	102 + 2	**151.66**	587.85	**10 392.94**	3 288.55
2	2	28 + 3	18.21	246.90	866.46	7 962.37
3	1	33 + 5	**54.72**	400.27	**14 310.20**	23 819.53
2	1	48 + 4	251.91	560.40	timeout	timeout
4	1	38 + 7	7 548.30	12 901.00	OOM	OOM
1	2	81 + 8	56 501.63	11 606.98	OOM	OOM
2	2	134 + 4	2 878.35	2 131.24	timeout	timeout
4	2	42 + 9	26 804.35	19 890.16	OOM	OOM
1	3	59 + 4	184.98	545.51	timeout	timeout
1	3	66 + 1	77.44	1 325.73	1 442.53	10 312.03
2	3	149 + 4	7 467.28	3 901.95	timeout	timeout
3	1	129 + 2	**769.27**	3 005.60	**151 403.08**	52 682.78
3	3	63 + 3	105.41	621.27	81 637.13	41 098.79
3	3	197 + 1	1 115.17	6 529.34	26 151.43	55 853.34
3	3	127 + 4	8 154.62	8 112.27	timeout	timeout
4	3	134 + 3	4 002.69	9 294.60	timeout	timeout
3	10	161 + 2	981.33	4 854.11	timeout	timeout
4	6	90 + 7	14 090.17	5 948.56	OOM	OOM
4	11	20 + 7	2 436.91	13 873.57	OOM	OOM
3	21	34 + 5	343.70	1 555.08	timeout	timeout

Table 1. Performance comparison of proposed algorithm to the algorithm of [25] on an equivalent DPN.

We used additional logical operators defined as follows: $x \vee y$ iff $\neg(\neg x \wedge \neg y)$ (*logical or*), and $x \implies y$ iff $\neg x \vee y$ (*implication*). Each standard LTL formula f_i is represented as $\left[f_i\right]^i$ to specify that this formula belongs to the process i.

Mirai malware is a botnet virus that targets IoT devices running on Linux. Our sample was built for machines with 32-bit ARM processors. It has two parallel processes. One process performs DDoS attacks and communicates with a Command and Control (C2) server. The other process evades detection and maintains persistence on the host machine. The single-indexed LTL formula for Mirai can be described as:

$$f = \Big[\mathbf{F}\big(accept \wedge \mathbf{F}fork\big)\Big]^1 \wedge \Big[\mathbf{F}\big(mount \wedge \mathbf{F}prctl\big)\Big]^2$$

Intuitively, the first process waits for a command from C2 server using the *accept* syscall and then creates a parallel process to perform DDoS attack using the $fork$ syscall. The second process disguises itself as a regular software. First, it calls *mount* to create a new filesystem to hide there. Then, it changes the name of itself to a legitimate name using *prctl* syscall.

Gozi malware is a Windows virus that targets banking field and steals credentials to critical systems such as banking software. The malware uses threads to run different jobs in parallel. For example, the first thread tries to hide the presence of the malware by manipulating the filesystem. The second thread waits for some time to avoid automatic detection by antivirus software, and then tries to connect to the C2 server or spawn another process. The third thread steals credentials from browsers and saves them in a temporary file. Here is the single-indexed formula for such behaviour:

$$\begin{aligned} f = \Big[\mathbf{F}\big(FindFirstFileW \wedge \mathbf{F}(DeleteFileW \vee SetFileAttributesW)\big)\Big]^1 \wedge \\ \Big[\mathbf{F}\big(Sleep \wedge \mathbf{F}(CreateProcessW \vee connect)\big)\Big]^2 \wedge \\ \Big[\mathbf{G}\big(GetWindowTextW \implies \mathbf{F}CreateFileW\big)\Big]^3 \end{aligned}$$

The first thread finds the file of itself using $FindFirstFileW$ and hides it by either deleting it with $DeleteFileW$, or making it hidden using $SetFileAttributesW$. The second thread waits until antivirus software marks the program as safe by calling *Sleep*. Then, the malware establishes connection with C2 using *connect* or calls another malicious process using $CreateProcessW$. The third thread performs credential stealing by calling $GetWindowTextW$ on a browser, and saving the credentials that user writes to login forms into a new file created by the syscall $CreateFileW$.

Generic Backdoor is a group of different malwares that allows a malicious party to obtain a full access to the infected machine. The obtained sample of the backdoor employs four threads. The first one modifies the system registry in such a way that the backdoor file is always executed at the startup of the system. The second thread establishes connection with a C2 server. The third

thread sends data to C2. And the fourth thread waits for commands from C2. The behaviour is dscribed using the following formula:

$$f = \Big[\mathbf{F}(GetModuleFileNameA \wedge \mathbf{F}RegSetValueExA)\Big)\Big]^1 \wedge$$
$$\Big[\mathbf{F}(gethostbyname \wedge \mathbf{F}recv)\Big]^2 \wedge \Big[\mathbf{F}send\Big]^3 \wedge \Big[\mathbf{GF}(accept \wedge \mathbf{F}CreateThread)\Big]^4$$

The first thread gets its own filename using $GetModuleFileNameA$, and puts it into registry for automatic execution on system startup using the system call $RegSetValueExA$. The second formula finds the C2 server using $gethostbyname$ and configures a socket to receive incoming messages with $recv$. The third thread sends data using $send$ syscall. And the fourth thread waits for incoming messages using $accept$ and, on receiving the message, spawns a new thread using $CreateThread$ to perform malicious activities.

8 Conclusion

In this work, we propose a *direct* and *efficient* algorithm for model checking of SM-DPNs over single-indexed LTL formulas. First, we show an algorithm for reducing model checking SM-DPDS to the reachability analysis of Self-Modifying Büchi Dynamic Pushdown Systems using Multi-Automata. Then, we give an algorithm for single-indexed LTL model checking by computing fixpoint of DCLICs. During the experiments, we compared our algorithm with an approach of translating SM-DPN into standard DPN and performing LTL model checking on DPN, and the results show the efficiency of our approach. Finally, we show how the model checking can be applied for specification of self-modifying and concurrent malware.

References

1. abuse.ch: Malwarebazaar (2025), `https://bazaar.abuse.ch/about/`
2. Àlvarez, S.: The Official Radare2 Book. `https://book.rada.re/credits/credits.html`, accessed: 2025-01-15
3. Anckaert, B., Madou, M., De Bosschere, K.: A model for self-modifying code. In: Information Hiding: 8th International Workshop, IH 2006, Alexandria, VA, USA, July 10-12, 2006. Revised Selcted Papers 8. pp. 232–248. Springer (2007)
4. Apt, K., De Boer, F.S., Olderog, E.R.: Verification of sequential and concurrent programs. Springer Science & Business Media (2010)
5. Blazy, S., Laporte, V., Pichardie, D.: Verified abstract interpretation techniques for disassembling low-level self-modifying code. Journal of Automated Reasoning **56**, 283–308 (2016)
6. Bonfante, G., Marion, J.Y., Reynaud-Plantey, D.: A computability perspective on self-modifying programs. In: 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods. pp. 231–239. IEEE (2009)

7. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: CONCUR'97: Concurrency Theory: 8th International Conference Warsaw, Poland, July 1–4, 1997 Proceedings 8. pp. 135–150. Springer (1997)
8. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR 2005–Concurrency Theory: 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005. Proceedings 16. pp. 473–487. Springer (2005)
9. Cai, H., Shao, Z., Vaynberg, A.: Certified self-modifying code. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 66–77 (2007)
10. Corvus Forensics: Virus share, `https://virusshare.com/about`
11. Dawei, S., Delong, L., Zhibin, Y.: Dynamic self-modifying code detection based on backward analysis. In: Proceedings of the 2018 10th International Conference on Computer and Automation Engineering. pp. 199–204 (2018)
12. Diaz, M., Touili, T.: Model checking dynamic pushdown networks with locks and priorities. In: Networked Systems: 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9–11, 2018, Revised Selected Papers 6. pp. 240–251. Springer (2019)
13. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: International Conference on Computer Aided Verification. pp. 232–247. Springer (2000)
14. Esparza, J., Kučera, A., Schwoon, S.: Model checking ltl with regular valuations for pushdown systems. Information and Computation **186**(2), 355–376 (2003)
15. Hex-Rays: Ida pro. `https://hex-rays.com/ida-pro`, accessed: 2025-01-15
16. Holzmann, G.J.: Logic verification of ansi-c code with spin. In: International SPIN Workshop on Model Checking of Software. pp. 131–147. Springer (2000)
17. Kahlon, V., Gupta, A.: An automata-theoretic approach for model checking threads for ltl propert. In: 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06). pp. 101–110. IEEE (2006)
18. Kidd, N., Lammich, P., Touili, T., Reps, T.: A decision procedure for detecting atomicity violations for communicating processes with locks. International Journal on Software Tools for Technology Transfer **13**(1), 37–60 (2011)
19. Kinder, J., Veith, H.: Jakstab: a static analysis platform for binaries: tool paper. In: Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20. pp. 423–427. Springer (2008)
20. Messahel, W., Touili, T.: Reachability analysis of concurrent self-modifying code. In: International Conference on Engineering of Complex Computer Systems. pp. 257–271. Springer (2024)
21. Nguyen, H.V., Touili, T.: Caret analysis of multithreaded programs. In: Logic-Based Program Synthesis and Transformation: 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017, Revised Selected Papers 27. pp. 73–90. Springer (2018)
22. Pommellet, A., Touili, T.: Ltl model checking for communicating concurrent programs. Innovations in Systems and Software Engineering **16**(2), 161–179 (2020)
23. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE symposium on security and privacy (SP). pp. 138–157. IEEE (2016)
24. Song, F., Touili, T.: Efficient malware detection using model-checking. In: International Symposium on Formal Methods. pp. 418–433. Springer (2012)

25. Song, F., Touili, T.: Model checking dynamic pushdown networks. Formal Aspects of Computing **27**, 397–421 (2015)
26. Touili, T., Ye, X.: Ltl model checking of self modifying code. Formal Methods in System Design **60**(2), 195–227 (2022)
27. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In: 2015 IEEE Symposium on Security and Privacy. pp. 659–673. IEEE (2015)
28. Vardi, M.Y., Wolper, P.: Automata theoretic techniques for modal logics of programs. In: Proceedings of the sixteenth annual acm symposium on theory of computing. pp. 446–456 (1984)

A Proof of Lemma 1

Proof (Proof of Lemma 1). We prove in two directions:
($pre^*(W) \subseteq \mathcal{L}(\mathcal{A}_i^{pre^*})$): In other words, we need to prove that for every $c = (\langle p, \omega\rangle\theta) \in Conf_i$ and $D \subseteq \mathcal{D}_i$, if $c \overset{D}{\Longrightarrow}^* c_0$ for some $(c_0, D_0) \in W$, then $(p,\theta) \xrightarrow{\omega/D\cup D_0}{}^* q_f$ for some $q_f \in Acc_i$. Let $c_0 = (\langle p_0, \omega_0\rangle\theta_0)$ and $q \xrightarrow{\omega}_j^* q'$ be a path using transitions from δ_i' after adding j new transitions for $j \geq 0$. Intuitively, $q \xrightarrow{\omega/D}_0^* q'$ means that there is a path in δ_i. Since $(c_0, D_0) \in W \implies (c_0, D_0) \in \mathcal{L}(\mathcal{A}_i) \implies (p_0, \theta_0) \xrightarrow{\omega_0/D_0}_0^* q_f$, where $q_f \in Acc_i$. Assume $c \overset{D}{\Longrightarrow}^k c'$. We proceed with induction on k.

Basis $k = 0$. $p_0 = p$, $\theta_0 = \theta$, $\omega_0 = \omega$, $D = \emptyset$. The proposition holds.

Step $k \geq 1$. Let $(\langle p, \omega\rangle\theta) \overset{D'}{\Longrightarrow}^1 (\langle p', \omega'\rangle\theta')$ and $(\langle p', \omega'\rangle\theta') \overset{D''}{\Longrightarrow}^{k-1} (\langle p_0, \omega_0\rangle\theta_0)$, $D = D' \cup D''$. From the induction hypothesis, there is a path $(p', \theta') \xrightarrow{\omega'/D''\cup D_0}{}^* q_f$. From the definition of $\overset{D}{\Longrightarrow}$, since $(\langle p, \omega\rangle\theta) \overset{D'}{\Longrightarrow} (\langle p', \theta'\rangle\theta')$, one of these propositions holds

1. This transition was caused by a rule $r = p\gamma \hookrightarrow p'v \in \theta$. Then, $\omega = \gamma u$, $\omega' = vu$, $D' = \emptyset$, and $\theta = \theta'$, where $\gamma \in \Gamma_i$, $u, v \in \Gamma^*$. Let $(p', \theta') \xrightarrow{v/D_1}{}^* q' \xrightarrow{u/D_2}{}^* q_f$, such that $D_1 \cup D_2 = D'' \cup D_0$. Then, rule μ_1 applies, and we add a transition $(p,\theta) \xrightarrow{\gamma/D_1} q'$. Thus, with the new transition, $(p,\theta) \xrightarrow{\gamma/D_1}{}^* q' \xrightarrow{u/D_2}{}^* q_f$. $D = D' \cup D_0 = D_0$ and therefore, $D_2 \cup D_1 = D'' \cup D_0 = D'' \cup D$.
2. This transition was caused by a rule $r = p\gamma \hookrightarrow p'v \triangleright p_2\omega_2\theta_2 \in \theta$. Then, $\omega = \gamma u$, $\omega' = vu$, $D' = \{p_2\omega_2\theta_2\}$, and $\theta = \theta'$, where $\gamma \in \Gamma_i$, $u, v \in \Gamma^*$. Let $(p', \theta') \xrightarrow{v/D_1}{}^* q' \xrightarrow{u/D_2}{}^* q_f$, such that $D_1 \cup D_2 = D'' \cup D_0$. Then, rule μ_2 applies, and we add a transition $(p,\theta) \xrightarrow{\gamma/D_1\cup\{p_2\omega_2\theta_2\}} q'$. Thus, with the new transition, $(p,\theta) \xrightarrow{\gamma/D_1\cup\{p_2\omega_2\theta_2\}}{}^* q' \xrightarrow{u/D_2}{}^* q_f$. $D = D' \cup D_0 = D_0 \cup \{p_2\omega_2\theta_2\}$ and therefore, $D_2 \cup D_1 \cup \{p_2\omega_2\theta_2\} = D'' \cup D_0 \cup \{p_2\omega_2\theta_2\} = D'' \cup D$.
3. This transition was caused by a rule $r = p \xhookrightarrow{(\rho_1,\rho_2)} p' \in \theta$. Then, $\omega = \omega'$, $D' = \emptyset$, $\rho_1 \subseteq \theta$, $\rho_2 \subseteq \theta'$, and $\theta' = (\theta \backslash \rho_1) \cup \rho_2$. Let $\omega = \gamma u$ for $\gamma \in \Gamma$ and $u \in \Gamma^*$.

Then, $(p', \theta') \xrightarrow{\gamma/D_1} q' \xrightarrow{u/D_2}{}^* q_f$, such that $D_1 \cup D_2 = D'' \cup D_0$. Then, rule μ_3 applies, and we add a transition $(p, \theta'') \xrightarrow{\gamma/D_1} q'$. $\theta'' = (\theta' \backslash \rho_2) \cup \rho_1 = \theta$. Thus, with the new transition, $(p, \theta) \xrightarrow{\gamma/D_1}{}^* q' \xrightarrow{u/D_2}{}^* q_f$. $D = D' \cup D_0 = D_0$ and therefore, $D_2 \cup D_1 = D'' \cup D_0 = D'' \cup D$.

Thus, the induction hypothesis holds for all three cases during its step, and the lemma holds for this direction.

($\mathcal{L}(\mathcal{A}_i^{pre^*}) \subseteq pre^*(W)$): We prove by induction with the following hypothesis:

If there is a path $(p, \theta) \xrightarrow{\omega/D}{}^* q$, then there exists a configuration $(\langle p', \omega' \rangle \theta')$, such that:

- $(\langle p, \omega \rangle \theta) \overset{D'}{\Longrightarrow}{}^* (\langle p', \omega' \rangle \theta')$, $(p', \theta') \xrightarrow{\omega'/D_0}{}_0^* q$, and $D = D' \cup D_0$, and
- if q is an initial state, then $\omega' = \varepsilon$, $D' = \emptyset$.

Let $(p, \theta) \xrightarrow{\omega/D}{}_k^* q$ for some k. We proceed with the induction on k:

Basis $k = 0$. Then, $p = p'$, $\theta = \theta'$, $\omega = \omega'$, and $D = D'$ and therefore, the first item holds. If q is an initial state, considering also that we excluded transitions into initial states for $\mathcal{A}_i$, then q must be (p, θ) and $q \xrightarrow{\varepsilon/\emptyset}{}^* q$.

Step $k \geq 1$. Let $t = (p_1, \theta_1) \xrightarrow{\gamma/D_1} q_1$ be the k-th transition added to δ_i'. Let j be the number of times t was used in the path $(p, \theta) \xrightarrow{\omega/D}{}_k^* q$. We proceed with induction on j:

Basis $j = 0$. Then, $(p, \theta) \xrightarrow{\omega/D}{}_{k-1}^* q$ and therefore, the lemma holds by applying the induction hypothesis on $k - 1$.

Step $j \geq 1$. Then, there exists a path $(p, \theta) \xrightarrow{u/D_2}{}_{k-1}^* (p_1, \theta_1) \xrightarrow{\gamma/D_1} q_1 \xrightarrow{v/D_3}{}_{k-1}^* q$, such that $\omega = u\gamma v$ and $D = D_1 \cup D_2 \cup D_3$. We apply the induction hypothesis on $(p, \theta) \xrightarrow{u/D_2}{}_{k-1}^* (p_1, \theta_1)$ to obtain that there exist $p'' \in P_i$, $\theta'' \subseteq 2^{\Delta_i \cup \Delta_i^c}$, $D'' \subseteq \mathcal{D}_i$, such that $(p'', \theta'') \xrightarrow{\omega''/D''}{}_0^* (p_1, \theta_1)$. Moreover, $(\langle p, u \rangle \theta) \overset{D_2 \cup D''}{\Longrightarrow}{}^* (\langle p'', \omega'' \rangle \theta'')$, and since (p_1, θ_1) is an initial state, we apply the second part of the induction hypothesis to get that $\omega'' = \varepsilon$ and $D'' = \emptyset$. Therefore, $(\langle p, u \rangle \theta) \overset{D_2}{\Longrightarrow}{}^* (\langle p_1, \varepsilon \rangle \theta_1)$

Now, we consider the transition $(p_1, \theta_1) \xrightarrow{\gamma/D_1} q_1$. Because it was added by a saturation rule, one of these cases must hold:

Case μ_1: There exists a rule $r = p_1\gamma \hookrightarrow p'''\omega''' \in \theta_1$ for some $p''' \in P_i$ and $\omega''' \in \Gamma_i^*$, and $(p''', \theta_1) \xrightarrow{\omega'''/D_1}{}_{k-1}^* q_1$. The rule r allows the successor relation $(\langle p_1, \gamma \rangle \theta_1) \overset{\emptyset}{\Longrightarrow} (\langle p''', \omega''' \rangle, \theta_1)$ to hold. Now, we can extend this path backwards: $(\langle p, u\gamma \rangle \theta) \overset{D_2}{\Longrightarrow}{}^* (\langle p_1, \gamma \rangle \theta_1) \overset{\emptyset}{\Longrightarrow} (\langle p''', \omega''' \rangle \theta_1)$. Since $(p''', \theta_1) \xrightarrow{\omega'''/D_1}{}_{k-1}^* q_1 \xrightarrow{v/D_3}{}_q^*$ uses transition t fewer times, we apply the induction hypothesis to obtain that there exists a configuration $(\langle p_0, \omega_0 \rangle)\theta_0$, such that $(\langle p''', \omega''' \rangle \theta_1) \overset{D_4}{\Longrightarrow}{}^*$

$(\langle p_0, \omega_0\rangle\theta_0)$, $(p_0, \theta_0) \xrightarrow{\omega_0/D_5}_0^* q$, and $D_1 \cup D_3 = D_4 \cup D_5$. Now, we get $(\langle p, u\gamma\omega'''\rangle\theta) \overset{D_2}{\Longrightarrow}^*$ $(\langle p''', \omega'''\rangle\theta_1) \overset{D_4}{\Longrightarrow}^* (\langle p_0, \omega_0\rangle\theta_0)$. Now, we check the sets of DCLICs $D_2 \cup D_4 \cup D_5 = D_2 \cup D_1 \cup D_3 = D$. If q is an initial state and there are no transitions into initial states in $\mathcal{A}_i$, that means $q = (p_0, \theta_0)$, $\omega_0 = \varepsilon$, and $D_5 = \emptyset$. Thus, $D = D_2 \cup D_4$. Therefore, the induction hypothesis holds for this case.

Case μ_2: There exists a rule $r = p_1\gamma \hookrightarrow p'''\omega'''\triangleright p_2\omega_2\theta_2 \in \theta_1$ for some $p''' \in P_i$ and $\omega''' \in \Gamma_i^*$, and $(p''', \theta_1) \xrightarrow{\omega'''/D_6}_{k-1}^* q_1$, such that $D_1 = D_6 \cup \{p_2\omega_2\theta_2\}$. The rule r allows the successor relation $(\langle p_1, \gamma\rangle\theta_1) \overset{\{p_2\omega_2\theta_2\}}{\Longrightarrow} (\langle p''', \omega'''\rangle, \theta_1)$ to hold. Now, we can extend this path backwards: $(\langle p, u\gamma\rangle\theta) \overset{D_2}{\Longrightarrow}^* (\langle p_1, \gamma\rangle\theta_1) \overset{\{p_2\omega_2\theta_2\}}{\Longrightarrow}$ $(\langle p''', \omega'''\rangle\theta_1)$. Since $(p''', \theta_1) \xrightarrow{\omega'''/D_6}_{k-1}^* q_1 \xrightarrow{v/D_3}^* q$ uses transition t fewer times, we apply the induction hypothesis to obtain that there exists a configuration $(\langle p_0, \omega_0\rangle)\theta_0$, such that $(\langle p''', \omega'''\rangle\theta_1) \overset{D_4}{\Longrightarrow}^* (\langle p_0, \omega_0\rangle\theta_0)$, $(p_0, \theta_0) \xrightarrow{\omega_0/D_5}_0^* q$, and $D_6 \cup D_3 = D_4 \cup D_5$. Now, we get $(\langle p, u\gamma\omega'''\rangle\theta) \overset{D_2\cup\{p_2\omega_2\theta_2\}}{\Longrightarrow}^* (\langle p''', \omega'''\rangle\theta_1) \overset{D_4}{\Longrightarrow}^*$ $(\langle p_0, \omega_0\rangle\theta_0)$. Now, we check the sets of DCLICs $D_2 \cup \{p_2\omega_2\theta_2\} \cup D_4 \cup D_5 = D_2 \cup \{p_2\omega_2\theta_2\} \cup D_6 \cup D_3 = D_2 \cup D_1 \cup D_3 = D$. If q is an initial state and there are no transitions into initial states in $\mathcal{A}_i$, that means $q = (p_0, \theta_0)$, $\omega_0 = \varepsilon$, and $D_5 = \emptyset$. Thus, $D = D_2 \cup D_4$. Therefore, the induction hypothesis holds for this case.

Case μ_3: There exists $\theta''' \subseteq \Delta_i \cup \Delta_i^c$, such that there is a rule $r = p_1 \overset{(\rho_1,\rho_2)}{\hookrightarrow}$ $p''' \in \theta'''$, $\rho_2 \subseteq \theta$, such that $(p''', \theta''') \xrightarrow{\gamma/D_1}_{k-1}^* q_1$, and $\theta''' = (\theta_1 \backslash \rho_1) \cup \rho_2$ for some $p''' \in P_i$. According to the successor relation definition, $(\langle p_1, \omega'''\rangle\theta_1) \overset{\emptyset}{\Longrightarrow}$ $(\langle p''', \omega'''\rangle\theta''')$ for any $\omega''' \in \Gamma_i^*$. Thus, $(\langle p, u\omega'''\rangle\theta) \overset{D_2}{\Longrightarrow}^* (\langle p''', \omega'''\rangle\theta''')$. Since $(p''', \theta''') \xrightarrow{\gamma/D_1}^* q_1 \xrightarrow{D_3}^* q$ uses t fewer times, we apply the induction hypothesis to get that there exists a configuration $(\langle p_0, \omega_0\rangle)\theta_0$, such that $(\langle p''', \omega'''\rangle\theta''') \overset{D_4}{\Longrightarrow}^*$ $(\langle p_0, \omega_0\rangle\theta_0)$, $(p_0, \theta_0) \xrightarrow{\omega_0/D_5}_0^* q$, and $D_1 \cup D_3 = D_4 \cup D_5$. Then, $D_4 \cup D_5 \cup D_2 = D_1 \cup D_2 \cup D_3 = D$. If q is an initial state and there are no transitions into initial states in $\mathcal{A}_i$, that means $q = (p_0, \theta_0)$, $\omega_0 = \varepsilon$, and $D_5 = \emptyset$. Thus, $D = D_2 \cup D_4$. Therefore, the induction hypothesis holds for this case.

Finally, by proving the induction, we can apply this hypothesis to any transition $(p, \theta) \xrightarrow{\omega/D}^* q_f$, where $q_f \in Acc_i$, which means that $(\langle p, \omega\rangle\theta, D) \in \mathcal{L}(\mathcal{A}_i^{pre^*})$. Thus, we get that there is $(\langle p', \omega'\rangle\theta')$, such that $(\langle p, \omega\rangle\theta) \overset{D'}{\Longrightarrow}^* (\langle p', \omega'\rangle\theta')$, $(p', \theta') \xrightarrow{\omega'/D_0}_0^* q_f$, and $D = D' \cup D_0$. The path $(p', \theta') \xrightarrow{\omega/D_0}_0^* q_f$ means that $(\langle p', \omega'\rangle\theta', D_0) \in \mathcal{L}(\mathcal{A}_i)$, or $(\langle p', \omega'\rangle\theta', D_0) \in W$. The successor relation implies that $(\langle p, \omega\rangle\theta, D' \cup D_0) \in pre^*(W)$. Thus, any configuration with a set of DCLICs accepted by $\mathcal{A}_i^{pre^*}$ is a predecessor of some configuration with DCLICs in W.

By proving these two directions, we get that the set of configurations and DCLICs accepted by $\mathcal{L}(\mathcal{A}_i^{pre^*})$ is the same set as the set of predecessors of W.

B Proof of Theorem 2

Proof (Proof of Theorem 2). ($\Longrightarrow$): Let $\sigma = c_0 c_1 c_2 \dots$ is an accepting run of $\mathcal{BP}_i$, s.t. for $i \geq 0$, $c_i \overset{I_i}{\Longrightarrow} c_{i+1}$ and $D = \bigcup_i I_i$ is created during this run. D is finite because $\mathcal{D}_i$ is finite. Let $c_j = (\langle p_j, \omega_j \rangle \theta_j)$ for $j \geq 0$. Because we have a lower bound on the size of stack, we can construct a subsequence $c_{k_1} c_{k_2} \dots$, s.t. $|\omega_{k_1}| = min\{|\omega_j| \mid j \geq 0\}$ and $|\omega_{k_l}| = min\{|\omega_j| \mid j \geq k_{l-1}\}, l \geq 1$, where ω_m is a stack content of c_m for some m.

Hence, after c_{k_1} is reached, if $\omega_{k_1} = \gamma v$, then, for every $m > k_1$, $\omega_m = \omega'_m v$, where $\gamma \in \Gamma_i, v, \omega'_m \in \Gamma_i^*$. Moreover, since the number of states, phases, and transitions is limited, we can find a subsequence $c_{j_1} c_{j_2} \dots$, s.t. $p_{j_l} = p_0$, $\theta_{j_l} = \theta_0$, and $\omega_{j_l} = \gamma_0 \omega'_{j_l}$ for $l \geq 0$, where p_{j_l} and θ_{j_l} are state and phase at c_{j_l}. Therefore, the run uses transitions

$$c_0 \overset{D_4}{\Longrightarrow}{}^* \; c_{j_1} \overset{D_5}{\Longrightarrow}{}^+ \; c_g \overset{D_6}{\Longrightarrow}{}^*_{c_{j_m}}$$

, where $c_g = (\langle p_g \omega_g \rangle \theta_g)$, $p_g \in F_i$, $D_4 = \bigcup_{h=0}^{j_1 - 1} I_h$, $D_5 = \bigcup_{h=j_1}^{g-1} I_h$, $D_6 = \bigcup_{h=g}^{j_m - 1} I_h$, and $\forall h \geq j_m : I_h \subseteq D_6$.

Let $c_{j_1} = (\langle p', \gamma' \omega' \rangle \theta')$, then α_1 holds. Because c_{j_1} has the smallest stack for the run, ω' never changes afterwards, $\exists u, v \in \Gamma_i^*$, s.t. $\omega_g = u\omega'$ and $\omega_{j_m} = v\omega'$. Therefore, $(\langle p', \gamma' \rangle \theta') \overset{D_5}{\Longrightarrow}{}^+$ and $(\langle p_g, u \rangle \theta_g) \overset{D_6}{\Longrightarrow}{}^* (\langle p', \gamma' v \rangle) \theta'$.

($\Longleftarrow$) from α_2, we can construct a run $(\langle p_o, \gamma_0 v^k \omega \rangle \theta_0) \overset{D_2}{\Longrightarrow}{}^+ (\langle p_g, u v^k \omega \rangle \theta_g)$ and $(\langle p_g u v^k \rangle \theta_g) \overset{D_3}{\Longrightarrow}{}^* (\langle p_0 \gamma_0 v^{k+1} \omega \rangle \omega)$ for every $k \geq 0$. Since $(p_g, \theta_g) \in F_i$, then the run is accepting.

C Proof of Theorem 4

Proof (Proof of Theorem 4). We prove in both directions.

($\Longrightarrow$): Suppose the accepting global run ρ, s.t. $\forall \sigma \in \rho : \sigma \models f_{\pi(\sigma)}$. Let some σ_c be a local run starting at $c = (\langle p, \omega \rangle \theta)$. Therefore, $\exists D_c \in \mathcal{D}_{\pi(p)}$, s.t. $c \models_{D_c} f_{\pi(c)}$, or $(c, D_c) \in \mathcal{L}(\mathcal{A}_{\pi(c)})$ and $\forall (p' \omega' \theta') \in D_c : (\langle p', \omega' \rangle \theta') \models f_{\pi(p')}$. Let D_j be the j-th iteration of $F(D)$. We need to show that $c \in D_j$ for every j. We proceed by induction on j:

Basis $j = 0$. $D_j = D_I \implies p\omega\theta \in D_j$ by definition of D_I.

Step $j \geq 1$. By definition of $F(D)$, $D_j = \{(p\omega\theta) \in D_I \mid \exists D' \subseteq D_{j-1} : (\langle [p, g^0_{\pi(p)}], \omega \rangle \theta, D) \in \mathcal{A}_{\pi(p)}\}$. By applying the induction hypothesis on $(\langle p', \omega' \rangle \theta') \models f_{\pi(p')}$, we get that $(\langle p', \omega' \rangle \theta') \in D_{j-1}$, and thus, $(\langle p, \omega \rangle \theta) \in \mathcal{D}_{fp}$.

($\Longleftarrow$): By the definition of the set $\mathcal{D}_{fp}$ and the function F, the presence of a configuration c in $\mathcal{D}_{fp}$ requires $(c, D_c) \in \mathcal{L}(\mathcal{A}_{\pi(c)})$, or in other words, that $c \models_{D_c} f_{\pi(c)}$. for some D_c.