SMODIC: A Model Checker for Self-modifying Code

Tayssir Touili, Xin Ye

SMODIC: A Model Checker for Self-modifying Code

Tayssir Touili
CNRS, LIPN and University Paris 13
France

Xin Ye
fortiss- Research Institute of the Free State of Bavaria
Germany

ABSTRACT

In this paper, we present SMODIC, a model checker for self-modifying binary codes. SMODIC uses Self Modifying Pushdown Systems (SM-PDS) to model self-modifying binary code. This allows to faithfully represent the program's stack as well as the self-modifying instructions of the program. SMODIC takes a self-modifying binary code or a self modifying pushdown system as input. It can then perform reachability analysis and LTL/CTL model-checking for these models. We successfully used SMODIC to model-check more than 900 self-modifying binary codes. In particular, we applied SMODIC for malware detection, since malwares usually use self-modifying instructions, and since malicious behaviors can be described by LTL or CTL formulas. In our experiments, SMODIC was able to detect 895 malwares and to prove that 200 benign programs were benign. SMODIC was also able to detect several malwares that well-known antiviruses such as Bit-Defender, Kinsoft, Avira, eScan, Kaspersky, Baidu, Avast, and Symantec failed to detect. SMODIC can be found in https://lipn.univ-paris13.fr/~touili/smodic

CCS CONCEPTS

• **Theory of computation → Verification by model checking**;
• **Security and privacy → Logic and verification**.

KEYWORDS

Malware detection, Pushdown Systems, Model Checking

ACM Reference Format:
Tayssir Touili and Xin Ye. 2018. SMODIC: A Model Checker for Self-modifying Code. In *ARES2022*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Self-modifying code is code that modifies its own instructions while it is executing. It has been used for a long time to hide the internals of a program. It was e.g. applied to reverse engineering for protection [29], since hiding the codes of a program can protect some intellectual property contained by software. Recently, it has also been widely used by malware writers to hide their malicious intent and evade from anti-virus detection. As malwares have become a big security threat to our daily life, malware detection is a critical problem in both industry and academic areas. Thus, being able to analyse self-modifying code is becoming of the utmost importance, since it is widely used by malwares.

There are several kinds of possibilities to make a binary code self-modifying. One of these techniques is packing and unpacking. Such techniques were extensively studied e.g. in [4, 6, 10, 15, 21, 33]. In this work, we consider self-modifying code implemented by **self-modifying instructions**, which are instructions that consider code as data. This allows them to read and write into code, thus producing self-modifying instructions. Such instructions are usually **mov** instructions, since they allow to read and write into memory. To illustrate this, let us consider the code of Figure 1. This is a segment of the worm Worm.Whboy equipped with a self-modifying instruction. The goal of this worm is to spread itself and infect computers through file downloading. The first step for Worm.Whboy to infect a host is to make a copy of itself into it. For this, it needs to call the API function GetSystemDirectoryA (address `0x13`) to get its location, and then the API functions LStrCatN and CheckPath (addresses `0x2a` and `0x35`) to check the path. Let us now show how self-modifying code can fool a static analyser and can make this malware undetectable by an antivirus. In Figure 1, the box on the left gives, respectively, the binary code, the corresponding addresses of the different instructions, and the corresponding assembly instruction at each address. For example, `ff` is the binary code of the instruction `push`. Thus, the first line is translated to `push 0b`. The second line is translated to `mov 0x2 0xc`, since `c6` is the binary code of the instruction `mov`, etc. Let us now execute this code. First, `push 0b` is executed, then `mov 0x2 0xc`. This last instruction will replace the first byte at address `0x2` by `0xc`. Thus, at address `0x2`, `ff 0b` is replaced by `0c 0b`. Since `0c` is the binary code of `jmp`, this means the instruction `push 0b` is replaced by `jmp 0xb`. Therefore, this code is self-modifying. If we model this piece of code blindly, without looking at the semantics of the different instructions, we will extract from it the Control Flow Graph `CFG a` of Figure 1, in which the API functions GetSystemDirectoryA, LStrCatN and CheckPath responsible of the malicious behavior cannot be reached. However, the *correct* Control Flow Graph of this piece of code is `CFG b`. Thus, if we do not take into account the fact that the instruction `mov 0x2 0xc` is self-modifying, then this code will be declared as benign, whereas it is malicious. Therefore, it is very important to be able to deal with self-modifying code implemented by **mov** instructions.

In this paper, we present SMODIC, a model checker for self-modifying binary code that use self-modifying **mov** instructions. In SMODIC, such binary code is modeled using Self Modifying Pushdown Systems (SM-PDS) [42], which is an extension of standard Pushdown Systems (PDS) that can modify its own instructions during its execution. As advocated in [42], using SM-PDSs is suitable for this kind of self-modifying binary code as it allows to faithfully represent the program's stack as well as the self-modifying **mov**

17th International Conference on Availability, Reliability and Security., Aug 23–26, 2022, Vienna

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

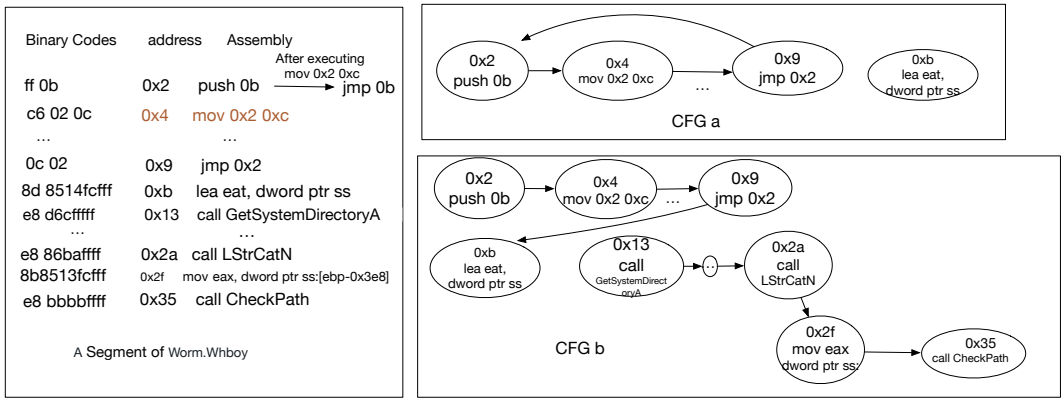

Figure 1: An Example of Self-modifying code

instructions of the program. SMODIC takes as input either a self-modifying binary code or a self modifying pushdown system. It can then perform reachability analysis and LTL/CTL model-checking for these models. SMODIC first adapts the tool Jakstab [22] to get the Control Flow Graph from the binary code. Then, it translates this CFG into a SM-PDS. It then implements the algorithms of [42, 43] to perform reachability analysis and LTL/CTL model-checking for this model.

We successfully used SMODIC to model-check more than 900 self-modifying binary codes. In particular, we applied SMODIC for malware detection, since malwares usually use self-modifying instructions. Indeed, malicious behaviors can be described by LTL or CTL formulas. In our experiments, SMODIC was able to detect 895 malwares and to prove that 200 benign programs were benign. SMODIC was also able to detect several malwares that well-known antiviruses such as Bit-Defender, Kinsoft, Avira, eScan, Kaspersky, Baidu, Avast, and Symantec failed to detect. SMODIC can be found in https://lipn.univ-paris13.fr/~touili/smodic.

Outline. Section 3 introduces the theoretical background of SMODIC. Section 4 shows how can SMODIC be used for malware detection. Section 5 describes the architecture of SMODIC. The experiments are discussed in Section 6.

2 RELATED WORK

Model checking and static analysis approaches have been widely used to analyze binary programs, for instance, in [4, 13, 15, 21, 33]. Temporal Logics were chosen to describe malicious behaviors in [13, 14, 19, 21, 30]. However, these works cannot deal with self-modifying code.

There are several researches on self-modifying code. Cai et al. [18] use local reasoning and separation logic to describe self-modifying code and treat program code uniformly as regular data structure. However, [18] requires programs to be manually annotated with invariants. In [36], the authors propose a formal semantics for self-modifying codes, and use that to represent self-unpacking code. This work only deals with packing and unpacking behaviours. Bonfante et al. [16] provide an operational semantics for self-modifying programs and show that they can be constructively rewritten to a non-modifying program. However, all these specifications [16, 18, 36] are too abstract to be used in practice. In [1], the authors propose a new representation of self-modifying code named State Enhanced-Control Flow Graph (SE-CFG). SE-CFG extends standard control flow graphs with a new data structure, keeping track of the possible states programs can reach, and with edges that can be conditional on the state of the target memory location. It is not easy to analyse a binary program only using its SE-CFG, especially that this representation does not allow to take into account the stack of the program. [34] propose abstract interpretation techniques to compute an over-approximation of the set of reachable states of a self-modifying program, where for each control point of the program, an over-approximation of the memory state at this control point is provided. [28] combine static and dynamic analysis techniques to analyse self-modifying programs. Unlike our approach, these techniques [28, 34] cannot handle the program's stack.

Unpacking binary code is also considered in [12, 23, 27, 32, 36]. These works do not consider self-modifying **mov** instructions.

There are a lot of tools that can deal with binary code analysis [2–4, 7–9, 11, 13, 14, 22, 24–26, 37, 39, 41]. POMMADE [13, 14] is a malware detector based on LTL and CTL model-checking of pushdown systems. STAMAD [24–26] is a malware detector based on PDSs and machine learning. However, all these tools cannot handle self-modifying code. The only tools that we know of and that can deal with self-modifying code are BE-PUM [17] and CoDisasm [5]. BE-PUM (Binary Emulation for PUshdown Model) [17] focuses on generating CFG (Control Flow Graph) of malwares. BE-PUM can construct a pushdown model from x86 binaries in an on-the-fly manner. Concolic testing is applied to determine the precise destinations of branches for indirect jumps. This tool can deal with self-modifying code caused by modifying the destinations of indirect jumps, including overwriting the return address of a function (in the stack). But it cannot handle self-modifying instructions. CoDisasm [5] is a tool that focuses on the disassembly of x86 code that includes self-modifying instructions and code overlapping. CoDisasm deals only with disassembling the code. It does not consider model-checking problems of code.

3 BACKGROUND

3.1 Self-modifying Pushdown Systems

A Self-modifying Pushdown System (SM-PDS) is a Pushdown System (PDS), i.e. an automaton equipped with a stack, that can dynamically modify its set of rules during the execution time: in addition to the standard push and pop transition rules of a pushdown system, a SM-PDS has self-modifying rules of the form $p \xhookrightarrow{(r_1,r_2)} p'$ that move the SM-PDS from control point p to control point p', while removing r_1 from the set of rules of the SM-PDS and adding r_2 to it. Thus, an SM-PDS has the capability to change its own set of transition rules during the execution. For more details about the SM-PDS model, we refer the reader to [42].

3.2 From Self-modifying Code to SM-PDS

To translate a binary code with self-modifying **mov** instructions to a SM-PDS, we use the translation of [42]. The basic idea is that the control locations of the SM-PDS store the control points of the binary program and the stack mimics the program's stack. Our translation relies on the disassembler Jakstab [22] to disassemble binary code, construct the control flow graph (CFG), determine indirect jumps, compute the possible values of the used variables, registers and the memory locations at each control point of the program. After getting the control flow graph whose edges are equipped with disassembled instructions, we translate the CFG into a SM-PDS as described in [42]. The non self-modifying instructions of the program define the standard PDS rules of the SM-PDS and can be obtained following the translation of [13] that models non self-modifying instructions of the program by a PDS. Self-modifying instructions are represented using self-modifying rules. For more details, we refer the reader to [42].

3.3 The Model Checking Algorithms

We use finite automata to finitely represent regular infinite sets of configurations of a SM-PDS. Then, the sets of predecessors and successors of a SM-PDS are computed using a kind of a saturation procedure on the finite automata [42]. As for LTL and CTL model-checking, they can be reduced to the emptiness problem for Self-modifying (Alternating) Büchi Pushdown Systems. The basic idea is to make a product of the given SM-PDS and the given LTL or CTL formula to get a Self-modifying (Alternating) Büchi Pushdown system and then apply the algorithms of [43] to check the emptiness of the Self-modifying (Alternating) Büchi Pushdown system.

4 APPLYING SMODIC FOR MALWARE DETECTION

SMODIC can be used for malware detection. Indeed, as shown in [13, 14], malicious behaviors can be described by LTL or CTL formulas. For example, we show in this section how LTL can be used to express the malicious behavior of registry key injecting. Other LTL and CTL formulas that describe other malicious behaviors can be found in [13, 14, 43].

In order to get started at boot time, many malwares add themselves into the registry key listing. This behavior is typically implemented by first calling the API function GetModuleFileNameA to retrieve the path of the malware's executable file. Then, the API function RegSetValueExA is called to add the file path into the registry key listing. As a result, the malware can execute itself automatically. This behavior can be expressed by the following LTL formula:

$$\mathbf{F}(call\ GetModuleFileNameA \wedge \mathbf{F}(\ call\ RegSetValueExA))$$

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

LdPinch.ch is such a malware that uses registry key injecting to execute itself. Fig. 3 shows a disassembled fragment of this malware. Note that this fragment contains self-modifying codes in address `00401547`: `mov` replaces the value at address `0040154A` with `6A` (which is the binary code of `push`). If we abstract away the fact that this code is self-modifying, and we treat this piece of code as if it were not self-modifying (as done in most of the other tools for binary code), then we will reach the conclusion that this code is not malicious. Indeed, in this case, a loop between address `00401545` and address `0040154A` will be detected and we will reach the conclusion that the API functions responsible of the malicious behavior cannot be called.

However, in reality, after the execution of `mov 0040154A c6`, the instruction at address `0040154A` will be changed to `push 00401545` and the remaining instructions will be executed: `GetModuleFileNameA` and `RegSetValueExA` will be called, and we will reach the conclusion that this code is malicious.

5 ARCHITECTURE

The Architecture of SMODIC is shown in Figure 2. SMODIC takes as input either a binary program or a SM-PDS. SMODIC can perform both reachability analysis and LTL/CTL model checking. If the input of SMODIC is a binary program, it is passed to the component **Oracle**. This component is based on the disassembler Jakstab [22]. It takes as input a binary program, and outputs its corresponding assembly program, its corresponding Control Flow Graph (CFG) equipped with the assembly instruction corresponding to each edge, together with informations about the called API functions, and the different values of the registers and memory addresses at each control point. All these outputs are fed to the component **Model Builder** that will compute the corresponding SM-PDS.

The **Reachability** component takes as input a SM-PDS, and a sequence of API functions, and applies the reachability algorithms of [42] to check whether the SM-PDS has a run that calls these API functions in this order. For example, if we consider the sequence f_1, f_2, f_3, then **Reachability** component checks whether the SM-PDS has a run that calls first f_1, then f_2, then f_3. The **LTL** (**CTL**) component takes as input a SM-PDS and an LTL (CTL) formula, and applies the algorithms of [43] to check whether the SM-PDS satisfies the LTL (CTL) formula.

6 EXPERIMENTS

A SM-PDS can be translated into an equivalent Pushdown System (PDS) [42]. Thus, to perform reachability, and LTL/CTL model-checking for an SM-PDS, one can translate it into an equivalent

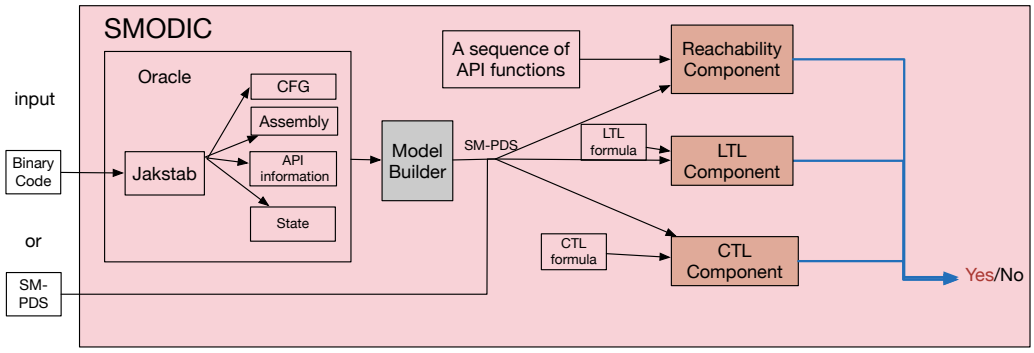

Figure 2: Architecture of SMODIC

```
   Address      Binary          Assembly
...
00401545   6A 0040154C       push 0040154C
00401547   C6 0040154A FF    mov 0040154A C6
0040154A   FF 00401545       jmp  00401545
0040154C    6A 00            push 0
0040154E   E8 47040000       call GetModuleFileNameA
...
004016E6   E8 63030000       call RegOpenKeyExA
...
004016F3   8365 F8 00        and dword ptr ss:[ebp-8],0
004016F7   6A 04             push 4
004016F9   8D45 F8           lea eax,dword ptr ss:[ebp-8]
004016FC    50               push eax
004016FD    6A 04            push 4
004016FF    6A 00            push 0
00401701   68 6B214000       push 0040216B
00401706   FF75 FC           push dword ptr ss:[ebp-4]
00401709   E8 46030000       call RegSetValueExA
...
```

Figure 3: A Fragment of LdPinch.ch

PDS, and then apply the standard algorithms for reachability and LTL/CTL model-checking for standard PDSs [20, 38, 40]. We show in this section how, in our experiments, translating the SM-PDS into an equivalent PDS is not efficient and that our tool SMODIC behaves much better. We also show how our tool SMODIC can be successfuly applied for malware detection. All our experiments were run on Ubuntu 16.04 with a 2.7 GHz CPU, 2GB of memory.

6.1 SMODIC vs. standard PDS Model-checking

To compare the performance of SMODIC against the approach that consists in translating the SM-PDS into an equivalent PDS or symbolic PDS [20, 38] and then apply the standard $post^*$, pre^*, LTL and CTL algorithms for PDSs and symbolic PDSs [20, 38, 40], we first applied our tool on randomly generated SM-PDSs of various sizes. SMODIC was able to successfully handle **all** cases in only a few seconds. Then, we translated these SM-PDSs into equivalent PDSs or symbolic PDSs and run the tools MOPED [38] (for reachability and LTL model checking), or PuMoC [40] (for CTL model checking). Going through PDSs or symbolic PDSs is less efficient and leads to memory out in several cases, whereas SMODIC was able to deal with all the cases in only a few seconds. The results (CPU Execution time) for the LTL component are shown in Table 1.

In Table 1, **Column** *Size* is the size of the SM-PDS (S_1 for non self-modifying transitions and S_2 for self-modifying transitions). **Column** *LTL* gives the size of the transitions of the Büchi automaton generated from the LTL formula (using the tool LTL2BA[31]). **Column** *SMODIC* gives the cost of SMODIC. **Column** *PDS* shows the cost it takes to get the equivalent PDS from the SM-PDS. **Column** *Result* reports the cost it takes to run the LTL PDS model-checker Moped [38] for the PDS we got. **Column** *Total* is the total cost it takes to translate the SM-PDS into a PDS and then apply the standard LTL model checking algorithm of Moped (Total=PDS+Result). **Column** *Symbolic PDS* reports the cost it takes to get the equivalent Symbolic PDS from the SM-PDS. **Column** $Result_1$ is the cost to run the Symbolic PDS LTL model-checker Moped. **Column** $Total_1$ is the total cost it takes to translate the SM-PDS into a symbolic PDS and then apply the standard LTL model checking algorithm of Moped. You can see that SMODIC (**Column** *SMODIC*) is much more efficient than translating the SM-PDS to an equivalent (symbolic) PDS, and then run the standard LTL model-checker Moped. **Translating the SM-PDS to a standard PDS may take more than 20 days, whereas our tool SMODIC takes only a few seconds.** Moreover, since the obtained standard (symbolic) PDS is huge, Moped failed to handle several cases (the time limit that we set for Moped is 20 minutes), whereas SMODIC was able to deal with all the cases in only a few seconds.

6.2 Detecting Real Malwares

We applied SMODIC to detect several malwares. We consider 895 malwares from VX heaven [44], VirusShare [45], and MalShare [35]. We also choose 200 benign samples from Windows XP system (win32). We consider self-modifying versions of the malwares. In these versions, the malicious behaviors are unreachable if the semantics of the self-modifying instructions are not taken into account, i.e., if the self-modifying instructions are considered as "standard" instructions that do not modify the code, then the malicious behaviors cannot be reached. First, we abstract away the semantics of the self-modifying instructions and model such programs as standard PDSs (as in [13, 14]), and perform LTL/CTL

Size	LTL	SMODIC	PDS	Result	Total	Symbolic PDS	$Result_1$	$Total_1$
$S_1:5, S_2:2$	$\|\delta\|$:15	**0.07s**	0.09s	0.01s	0 .10s	0.08s	0.00s	0.08s
$S_1:5, S_2:3$	$\|\delta\|$:8	**0.06s**	0.08s	0.01s	0.09s	0.09s	0.00s	0.09s
$S_1:11, S_2:4$	$\|\delta\|$:8	**0.16s**	0.13s	0.05s	0.18s	0.10s	0.00s	0.10s
$S_1:5, S_2:3$	$\|\delta\|$:10	**0.06s**	0.15s	0.01s	0.16s	0.09s	0.00s	0.09s
$S_1:110, S_2:4$	$\|\delta\|$:8	**0.34s**	186.10s	0.79s	186.99s	0.35s	0.00s	0.35s
$S_1:255, S_2:8$	$\|\delta\|$:8	**0.39s**	281.02s	0.94s	281.96s	4.82s	0.05s	4.87s
$S_1:255, S_2:8$	$\|\delta\|$:10	**0.42s**	281.02s	0.97s	281.99s	4.82s	0.06s	4.88s
$S_1:110, S_2:4$	$\|\delta\|$:15	**0.28s**	186.10s	1.05s	187.15s	0.35s	0.06s	0.41s
$S_1:255, S_2:8$	$\|\delta\|$:15	**0.46s**	281.02s	1.92s	282.94s	4.82s	0.08s	4.90s
$S_1:110, S_2:4$	$\|\delta\|$:20	**0.37s**	186.10s	1.05s	187.15s	0.35s	0.06s	0.41s
$S_1:255, S_2:8$	$\|\delta\|$:20	**0.55s**	281.02s	1.97s	282.99s	4.82s	0.17s	4.99s
$S_1:255, S_2:8$	$\|\delta\|$:25	**0.59s**	281.02s	1.23s	282.99s	4.82s	0.24s	5.36s
$S_1:2059, S_2:7$	$\|\delta\|$:8	**0.86s**	19525.01s	20.71s	19545.72s	20.70s	error	-
$S_1:2059, S_2:9$	$\|\delta\|$:8	**1.49s**	19784.7s	79.12s	19863.32	128.12s	error	-
$S_1:2059, S_2:11$	$\|\delta\|$:8	**3.73s**	30011.67s	168.15s	30179.82s	261.07s	error	-
$S_1:2059, S_2:11$	$\|\delta\|$:28	**6.88s**	30011.67s	169.55s	30180.22s	261.07s	error	-
$S_1:3050, S_2:10$	$\|\delta\|$:8	**5.21s**	39101.57s	killed	-	438.27s	error	-
$S_1:3090, S_2:10$	$\|\delta\|$:8	**5.86s**	40083.07s	killed	-	438.69s	error	-
$S_1:3050, S_2:10$	$\|\delta\|$:20	**7.24s**	39101.57s	killed	-	438.27s	error	-
$S_1:3090, S_2:10$	$\|\delta\|$:30	**8.38s**	40083.07s	killed	-	438.69s	error	-
$S_1:3090, S_2:10$	$\|\delta\|$:25	**8.89s**	40083.07s	killed	-	438.69s	error	-
$S_1:4050, S_2:10$	$\|\delta\|$:8	**9.21s**	81408.91s	killed	-	699.19s	error	-
$S_1:4050, S_2:10$	$\|\delta\|$:28	**11.64s**	81408.91s	killed	-	699.19s	error	-
$S_1:4058, S_2:11$	$\|\delta\|$:8	**9.83s**	93843.37s	killed	-	802.07s	error	-
$S_1:4058, S_2:11$	$\|\delta\|$:25	**13.59s**	93843.37s	killed	-	802.07s	error	-
$S_1:5050, S_2:11$	$\|\delta\|$:8	**10.34s**	173943.37s	killed	-	921.16s	error	-
$S_1:5090, S_2:11$	$\|\delta\|$:8	**10.52s**	179993.54s	killed	-	929.32s	error	-
$S_1:5090, S_2:11$	$\|\delta\|$:10	**12.89s**	179993.54s	killed	-	929.32s	error	-
$S_1:6090, S_2:11$	$\|\delta\|$:8	**13.49s**	190293.64s	killed	-	1002.73s	error	-
$S_1:6090, S_2:11$	$\|\delta\|$:10	**15.81s**	190293.64s	killed	-	1002.73s	error	-
$S_1:6090, S_2:11$	$\|\delta\|$:40	**32.39s**	190293.64s	killed	-	1002.73s	error	-
$S_1:10150, S_2:12$	$\|\delta\|$:60	**97.56s**	2134633.28s	killed	-	1469.28s	error	-
$S_1:10150, S_2:12$	$\|\delta\|$:65	**105.89s**	2134633.28s	killed	-	1469.28s	error	-
$S_1:10150, S_2:16$	$\|\delta\|$:65	**134.45s**	2211008.82s	killed	-	3665.59s	error	-
$S_1:10180, S_2:16$	$\|\delta\|$:65	**175.29s**	2134643.52s	killed	-	3689.83s	error	-
$S_1:10180, S_2:16$	$\|\delta\|$:78	**214.36s**	2134643.52s	killed	-	3689.83s	error	-

Table 1: SMODIC vs. standard LTL for PDSs

SMODIC	McAfee	Norman	BitDefender	Kinsoft	Avira	eScan	Kaspersky	Qihoo360	Avast	Symantec
100%	27.6%	22.1%	33.1%	14.4%	28.3%	21.4%	56.2 %	35.9%	50.7%	77.9%

Table 2: SMODIC vs. well known anti-viruses

model-checking for PDSs to determine whether the programs contain any malicious behavior. In this case, *none* of the programs was declared as malicious. Then, we use SM-PDSs to model these programs, thus, taking self-modifying instructions into consideration: we use SMODIC to model these programs using SM-PDSs and to check whether these SM-PDSs satisfy any malicious LTL/CTL formula in our database. SMODIC was able to detect all malwares, and to calssify benign programs as benign.

6.3 Comparison with well-known antiviruses.

We also compare our tool against well-known and widely used antiviruses. In order to have a fair comparision, we need to consider new malwares, since anti-viruses know the signatures of all the *known* malwares. Thus, the challenge for anti-viruses is to detect *new* malwares. To this aim, we use the sophisticated malware generator NGVCK available at VX Heavens [44] to generate 200 new malwares. Then we obfuscate these malwares with self-modifying code. Then, we feed these malwares to SMODIC and to well-known

antiviruses such as BitDefender, Kinsoft, Avira, eScan, Kaspersky, Qihoo-360, Baidu, Avast, and Symantec to detect them. Our tool was able to detect all these programs as malicious, whereas none of the well-known antiviruses was able to detect all these malwares. Table 2 reports the detection rates of our tool and the well-known anti-viruses.

REFERENCES

[1] A.Bertrand, M.Matias, and D.Koen. 2006. A model for self-modifying code. In *International Workshop on Information Hiding*.

[2] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86-A Platform for Analyzing x86 Executables*. In *Compiler Construction: 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*, Vol. 3443. Springer, 250.

[3] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. 2011. The BINCOA framework for binary code analysis. In *International Conference on Computer Aided Verification*. Springer, 165–170.

[4] Jean Bergeron, Mourad Debbabi, Jules Desharnais, Mourad M Erhioui, Yvan Lavoie, Nadia Tawbi, et al. 2001. Static detection of malicious code in executable programs. *Int. J. of Req. Eng* 2001, 184-189 (2001), 79.

[5] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. 2015. CoDisasm: medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 745–756.

[6] Guillaume Bonfante, Hubert Godfroy, and Jean-Yves Marion. 2017. A construction of a self-modifiying language with a formal correction proof. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 99–106.

[7] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. 2007. *Bitscope: Automatically dissecting malicious binaries.* Technical Report. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon.

[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.

[9] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting self-mutating malware using control-flow graph matching. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 129–143.

[10] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. 2005. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE, 32–46.

[11] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 653–656.

[12] Saumya Debray and Jay Patel. 2010. Reverse engineering self-modifying code: Unpacker extraction. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 131–140.

[13] F.Song and T.Touili. 2012. Efficient Malware Detection Using Model-Checking. In *International Symposium on Formal Methods*.

[14] F.Song and T.Touili. 2013. LTL model-checking for malware detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[15] G.Balakrishnan, T.W. Reps, N.Kidd, A.Lal, J.Lim, et al. 2005. Model Checking x86 Executables with CodeSurfer/x86 and WPDS++. In *International Conference on Computer Aided Verification*.

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

[17] Nguyen Minh Hai, O Mizuhito, and Quan Thanh Tho. 2014. *Pushdown model generation of malware.* Technical Report. Technical report, Japan Advanced Institute of Science and Technology, Japan.

[18] H.Cai, Z.Shao, and A.Vaynberg. 2007. Certified self-modifying code. *ACM SIGPLAN Notices* 42, 6 (2007).

[19] H.Nguyen and T.Touili. 2017. CARET model checking for malware detection. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*.

[20] J.Esparza, D.Hansel, P.Rossmanith, and S.Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In *International Conference on Computer Aided Verification*.

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

[22] H.Veith J.Kinder. 2008. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*.

[23] K.Coogan, S.Debray, T.Kaochar, and G.Townsend. 2009. Automatic static unpacking of malware binaries. In *16th Working Conference on Reverse Engineering*.

[24] K.Dam and T.Touili. 2017. Malware Detection based on Graph Classification. In *International Conference on Information Systems Security and Privacy*.

[25] K.Dam and T.Touili. 2018. Learning Malware Using Generalized Graph Kernels. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*.

[26] K.Dam and T.Touili. 2018. Precise Extraction of Malicious Behaviors. In *IEEE 42nd Annual Computer Software and Applications Conference*.

[27] K.Gyung et al. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*.

[28] K.Roundy and B.Miller. 2010. Hybrid analysis and control of malware. In *International Workshop on Recent Advances in Intrusion Detectio*.

[29] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. 2006. Software protection through dynamic code mutation. In *Information Security Applications*. Springer.

[30] P.Beaucamps, I.Gnaedig, and J.Marion. 2010. Behavior Abstraction in Malware Analysis. In *Runtime Verification*.

[31] P.Gastin and D.Oddoux. 2001. Fast LTL to Büchi automata translation. In *International Conference on Computer Aided Verification*.

[32] P.Royal, M.Halpin, et al. 2006. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *22nd Annual Computer Security Applications Conference (ACSAC'06)*.

[33] P.Singh and A.Lakhotia. 2003. Static verification of worm and virus behavior in binary executables using model checking. In *IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop*.

[34] S.Blazy, V.Laporte, and D.Pichardie. 2016. Verified abstract interpretation techniques for disassembling low-level self-modifying code. *Journal of Automated Reasoning* 56, 3 (2016).

[35] S.Cutler. 2022. malshare. https://malshare.com.

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

[37] Axel Simon and Julian Kranz. 2014. The GDSL toolkit: Generating frontends for the analysis of machine code. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM, 7.

[38] S.Schwoon S.Kiefer and D.Suwimonteerabuth. 2011. Moped - A Model-Checker for Pushdown Systems. http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/.

[39] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 1–25.

[40] Fu Song and Tayssir Touili. 2012. PuMoC: a CTL model-checker for sequential programs. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 346–349.

[41] Aditya Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas Reps. 2010. Directed proof generation for machine code. In *International Conference on Computer Aided Verification*. Springer, 288–305.

[42] T.Touili and X.Ye. 2017. Reachability Analysis of Self Modifying Code. In *22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*.

[43] T.Touili and X.Ye. 2019. LTL Model Checking for Self Modifying Code. In *24th International Conference on Engineering of Complex Computer Systems (ICECCS)*.

[44] V.Heaven. 2017. V.Heavens. http://vxer.org/lib/.

[45] VirusShare. 2022. vXshare. https://virusshare.com.