Efficient Malware Detection Using Model-Checking

Fu Song and Tayssir Touili

LIAFA, CNRS and Univ. Paris Diderot, France.
E-mail: {song,touili}@liafa.jussieu.fr

Abstract. Over the past decade, malware costs more than $10 billion every year and the cost is still increasing. Classical signature-based and emulation-based methods are becoming insufficient, since malware writers can easily obfuscate existing malware such that new variants cannot be detected by these methods. Thus, it is important to have more robust techniques for malware detection. In our previous work [23], we proposed to use model-checking to identify malware. We used pushdown systems (PDSs) to model the program (this allows to keep track of the program's stack behavior), and we defined the SCTPL logic to specify the malicious behaviors, where SCTPL can be seen as an extension of the branching-time temporal logic CTL with variables, quantifiers, and predicates over the stack. Malware detection was then reduced to SCTPL model-checking of PDSs. However, in our previous work [23], the way we used SCTPL to specify malicious behaviors was not very precise. Indeed, we used the *names* of the registers and memory locations instead of their *values*. We show in this work how to sidestep this limitation and use precise SCTPL formulas that consider the *values* of the registers and memory locations to specify malware. Moreover, to make the detection procedure more efficient, we propose an abstraction that reduces drastically the size of the program model, and show that this abstraction preserves all SCTPL\X formulas, where SCTPL\X is a fragment of SCTPL that is sufficient to precisely characterize malware specifications. We implemented our techniques in a tool and applied it to *automatically* detect several malwares. The experimental results are encouraging.

1 Introduction

The number of malwares that produced incidents in 2010 is more than 1.5 billion [14]. A malware may bring serious damage, e.g., the worm MyDoom slowed down global internet access by ten percent in 2004 [5]. Thus, it is crucial to have efficient up-to-date virus detectors. Existing antivirus systems use various detection techniques to identify viruses such as (1) code emulation where the virus is executed in a virtual environment to get detected; or (2) signature detection, where a signature is a pattern of program code that characterizes the virus. A file is declared as a virus if it contains a sequence of binary code instructions that matches one of the known signatures. Each virus variant has its corresponding signature. These techniques have some limitations. Indeed, emulation based techniques can only check the program's behavior in a limited time interval. They cannot check what happens after the timeout. Thus, they might miss the viral behavior if it occurs after this time interval. As for signature based systems, it is very easy to virus developers to get around them. It suffices to apply obfuscation techniques to change the structure of the code while keeping the same functionality, so that the new version does not match the known signatures. Obfuscation techniques can consist of inserting dead code, substituting

instructions by equivalent ones, etc. Virus writers update their viruses frequently to make them undetectable by these antivirus systems.

Recently, to sidestep these limitations, model-checking techniques have been used for virus detection [9, 22, 11, 12, 17, 16, 18]. Such techniques allow to check the *behavior* (not the syntax) of the program without executing it. These works use finite state graphs as program model. Thus, they cannot accurately represent the program's stack. However, as shown in [21], being able to track the program's stack is very important for malware detection. For example, to check whether a program is malicious, anti-viruses start by identifying the system calls it makes. To evade these virus detectors, malware writers try to obfuscate the system calls by using pushes and jumps. Thus, it is important to be able to track the stack to detect such calls.

To this aim, we proposed in our previous work [23] a new approach for malware detection that consists in (1) Modeling the program using a Pushdown System (PDS). This allows to take into account the behavior of the stack. (2) Introducing a new logic, called SCTPL, to represent the malicious behavior. SCTPL can be seen as an extension of the branching-time temporal logic CTL with variables, quantifiers, and predicates over the stack. (3) And reducing the malware detection problem to the model-checking problem of PDSs against SCTPL formulas. Our techniques were implemented in a tool and applied to detect several viruses.

However, [23] still has some limitations: (1) The PDS corresponding to the program to be analyzed was generated by hand by the user. (2) Due to the high complexity of SCTPL model-checking, we were not able to check several examples (they run out of memory). (3) When specifying malicious behavior using SCTPL, we used formulas where the variables range over the names of the program's registers, not over their values. Thus, the specifications were not precise. To understand this last problem, let us consider the program of Figure 1(a). It corresponds to a critical fragment of the Email-worm Klez that shows the typical behavior of an email worm: it calls the API function *GetModuleFileNameA* with 0 as first parameter and an address a as second parameter[1]. This function will store the file name of the worm's own executable into the memory pointed by a, so that later, the worm can infect other files by copying this executable stored in the memory pointed by a into them.

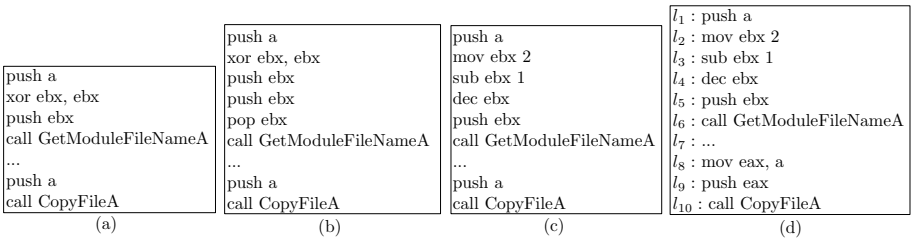

Fig. 1. (a) Worm fragment; (b), (c) and (d) Obfuscated fragments.

Using SCTPL, in [23] we specify this malicious behavior by the following formula:

[1] Parameters of a function in assembly are passed by pushing them into the stack before calling the function. The callee retrieves these parameters from the stack.

$$\psi = \exists a \exists r_1 \mathbf{EF}\Big(xor(r_1, r_1) \wedge \mathbf{EXE}[\neg\exists v\ mov(r_1, v)\mathbf{U}push(r_1) \wedge \mathbf{EXE}[\neg(push(r_1) \vee \exists r'(pop(r') \wedge r_1\Gamma^*))\mathbf{U}call(GetModuleFileNameA) \wedge r_1 a\Gamma^* \wedge \mathbf{EF}(call(CopyFileA) \wedge a\Gamma^*)]]\Big)$$

where $r_1 a\Gamma^*$ (resp. $a\Gamma^*$) is a regular predicate expressing that the topmost symbols of the stack are r_1 and a (resp. a). This SCTPL formula ψ states that there exists a register r_1 that is first assigned 0 ($xor(r_1, r_1)$) and such that it is not assigned any other value later until r_1 is pushed onto the stack. Later, r_1 is never popped from the stack nor pushed onto it again until the function *GetModuleFileNameA* is invoked. When this call is made, the topmost symbols of the stack have to be r_1 and a. This ensures that the first parameter of *GetModuleFileNameA* is the value of r_1, i.e. 0, and that the file name of its own executable returned by the function is stored in the memory pointed by a. This specification can detect the fragment in Figure 1(a). However, a virus writer can easily use some obfuscation techniques in order to escape from this specification. For example, if we add a *push ebx* followed by a *pop ebx* as done in Figure 1 (b); or instead of using *xor ebx ebx* to put 0 into *ebx*, let us put the value 2 in *ebx* and then remove 1 twice as done in Figure 1 (c). These two fragments keep the same malicious behavior than the fragment of Figure 1(a), however, they cannot be detected by the formula ψ. A virus writer can also escape from this specification by first assigning the address a to the register *eax* and then pushing the value of *eax* onto the stack as shown in Figure 1 (d) (instead of pushing a directly to the stack). When calling *CopyFileA*, the topmost symbol of the stack is equal to *the value* stored in a, but is different from *the name a*. Thus, this fragment cannot be detected by the above specification ψ.

To overcome this problem, we propose in this work to specify malicious behaviors using SCTPL formulas where the variables range over the values of the program's registers and memory, not over their names. In this way, the malicious behavior of Figures 1 (a), (b), (c) and (d) can be specified as follows:

$$\Omega = \exists m\ \mathbf{EF}(call(GetModuleFileNameA) \wedge \{0\}\ m\ \Gamma^* \wedge \mathbf{EF}(call(CopyFileA) \wedge m\Gamma^*))$$

This expresses that a call to the API function *GetModuleFileNameA* is made with 0 and the *value* of the address m of the memory on top of the stack, followed by a call to the API function *CopyFileA* with the *value* of m on top of the stack. Unlike [23], m represents the *values* of the program's registers and addresses, not their names.

In order to consider such specifications, we need to track the values of the different registers of the program. To do this, we consider an oracle $\mathcal{O}$ that gives an overapproximation of the current state at each control point of the program, i.e., an overapproximation of the values of the different registers and memory locations. To implement this oracle, we use Jakstab [19] and IDA Pro [6]. Based on the oracle $\mathcal{O}$, we implement a translator that *automatically* constructs a PDS from the binary program.

To overcome the high complexity problem of SCTPL model-checking, we introduce the *collapsing abstraction*, which is an abstraction that drastically reduces the size of the program model by removing the instructions that do not change the stack (instructions using push or pop are not removed), nor the control flow of the programs (instructions using jump-like operators, e.g., jmp, jz, etc. are not removed); as well as the instructions whose operators do not appear in the considered SCTPL formula. We show that this abstraction preserves all SCTPL\X formulas, where SCTPL\X is a subclass of SCTPL that uses the

next time operator X only to specify the return addresses of the callers. We show that this fragment SCTPL\X is sufficient to specify all the malicious behaviors we considered. Our abstraction allowed to apply our techniques to large programs. In our experiments, several examples terminate when we use our abstraction, whereas without it, they run out of memory.

The main contributions of this paper are:

1. We propose to specify malicious behaviors using SCTPL formulas where the variables range over the values of the program's registers, not over their names as done in [23]. Thus, we get more precise malware specifications.
2. We present a new approach to model a binary program as a PDS. Our translation is more precise than the other existing translations from programs to PDSs.
3. We identify the sub-logic SCTPL\X, which is a subclass of SCTPL where the next time operator X is used only to specify the return addresses of the callers. We show that SCTPL\X is sufficient to specify all the malware behaviors we considered, and we proposed the *collapsing abstraction* and show that it preserves SCTPL\X properties. This abstraction reduces drastically the model size, and thus makes the model-checking problem more efficient.
4. We implement our techniques in a tool for malware detection. All the steps are completely automated. Our tool takes as input a binary program and a set of SCTPL\X formulas representing a set of malicious behaviors. It outputs *"Yes, the program may be a malware"* if the program satisfies one of the formulas, and *"NO"* if not. We get encouraging results.

Related work. These last years, there has been a substantial amount of research to find efficient techniques that can detect viruses. A lot of techniques use signature based or emulation based approaches. As already mentioned in the introduction, such techniques have some limitations. Indeed, signature matching fails if the virus does not use a known signature. As for emulation techniques, they can execute the program only in a given time interval. Thus, they can miss the malicious behaviors if they occur after the timeout.

Model-checking and static analysis techniques have been applied to detect malicious behaviors e.g. in [9, 22, 11, 12, 17, 16, 18]. However, all these works are based on modeling the program as a finite-state system, and thus, they miss the behavior of the stack. As we have seen, being able to track the stack is important for many malicious behaviors. [10, 7] use tree automata to represent a set of malicious behaviors. However, these works cannot specify predicates over the stack content.

[21] keeps track of the stack by computing an abstract stack graph which finitely represents the infinite set of all the possible stacks for every control point of the program. Their technique can detect obfuscated calls and obfuscated returns. However, they cannot specify the other malicious behaviors that we are able to detect using our SCTPL specifications.

[20] performs context-sensitive analysis of *call* and *ret* obfuscated binaries. They use abstract interpretation to compute an abstraction of the stack. We believe that our techniques are more precise since we do not abstract the stack. Moreover, the techniques of [20] were only tried on toy examples, they have not been applied for malware detection.

[8] uses pushdown systems for binary program analysis. However, the translation from programs to PDSs in [8] assumes that the program follows a standard compilation model

where calls and returns match. As we have shown, several malicious behaviors do not follow this model. Our translation from a control flow graph to a PDS does not make this assumption.

SCTPL can be seen as an extension of CTPL with predicates over the stack content. CTPL was introduced in [17, 16, 18]. In these works, the authors show how CTPL can be used to succinctly specify malicious behaviors. Our SCTPL logic is more expressive than CTPL. Indeed, CTPL cannot specify predicates over the stack. Thus, SCTPL allows to specify more malicious behaviors than CTPL. Indeed, most of the malicious behaviors we considered cannot be expressed in CTPL.

Outline. In Section 2, we give our formal model. Section 3 recalls the definition of the SCTPL logic and shows how this logic can precisely represent malicious behavior. We give the definition of the fragment SCTPL\X and of the collapsing abstraction in Section 4. Our experiments are described in Section 5. Due to lack of space, proofs are omitted. They can be found in Appendix.

2 Binary Code Modeling

In this section, we show how to build a PDS from a binary program. We suppose we are given an oracle O that extracts from the binary program a control flow graph equipped with informations about the values of the registers and the memory locations at each control point of the program. In our implementation, we use Jakstab [19] and IDA Pro [6] to get this oracle. We translate the control flow graph into a pushdown system where the control locations store the control points of the binary program and the stack tracks the stack of the program. This translation takes into account the values of the different registers and memory locations of the program.

2.1 Control Flow Graphs

Let $\mathbf{R}$ be the finite set of registers used in the binary program. Let **States** be the set of functions from $\mathbf{R} \cup \mathbb{Z}$ to $2^{\mathbb{Z}}$ where $\mathbb{Z}$ is the set of integers. Intuitively, let $s \in$ **States**. For every $r \in \mathbf{R}$, $s(r)$ gives the possible values of the register r in the state s, while for every $d \in \mathbb{Z}$, $s(d)$ gives the possible values of the memory at address d in the state s. Let **EXP** be the set of expressions over the registers and the memory locations used in the program. **States** is extended over expressions in **EXP** in the usual way.

A *control flow graph* (CFG) is a tuple $G = (N, I, E)$, where N is a finite set of nodes corresponding to the control points of the program, I is a finite set of assembly instructions used in the program, and $E : N \times I \times N$ is a finite set of edges each of them associated with an assembly instruction of the program. We write $n_1 \xrightarrow{i} n_2$ for every (n_1, i, n_2) in E. Given a binary program, the oracle O computes a corresponding control flow graph G and a function $\varrho : N \longrightarrow$ **States** that associates to each node n an overapproximation of the set of possible states of the program at the control point n.

2.2 Pushdown Systems

A *Pushdown System* (PDS) is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules.

A configuration $\langle p, \omega \rangle$ of $\mathcal{P}$ is an element of $P \times \Gamma^*$. We write $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ instead of $((p, \gamma), (q, \omega)) \in \Delta$. The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$. A *path* of the PDS is a sequence of configurations $c_1, c_2, ...$ such that c_{i+1} is an *immediate successor* of the configuration c_i, i.e., $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$, for every $i \geq 1$.

2.3 From Control Flow Graphs to Pushdown Systems

In this section, we present a novel approach to derive a pushdown system from a control flow graph. Consider a binary program. Let (N, I, E) be the CFG and ϱ be the state function provided by the oracle $\mathcal{O}$. We construct the PDS $\mathcal{P} = (P, \Gamma, \Delta)$ as follows:

- the control locations P are the nodes N;
- Γ is the smallest set of symbols satisfying the following:
  - if $n \xrightarrow{call\ proc} n' \in E$, then $\{n'\} \in \Gamma$ where n' is the return address of the call;
  - if $n \xrightarrow{push\ exp} n' \in E$, where exp is an expression in EXP, then $\varrho(n)(exp) \in \Gamma$ where $\varrho(n)(exp)$ denotes the set of possible values of the expression exp at the control point n (given by the state $\varrho(n)$);
- the set of rules Δ contain transition rules that mimic the instructions of the program: for every edge $e \in E$, $\gamma \in \Gamma$:
  - if $e = n_1 \xrightarrow{push\ exp} n_2$, we add the transition rule $\langle n_1, \gamma \rangle \hookrightarrow \langle n_2, \gamma'\gamma \rangle$ where $\gamma' = \varrho(n_1)(exp)$. This rule moves the program's control point from n_1 to n_2, and pushes the set of all the possible values of the expression exp at control point n_1 onto the stack;
  - if $e = n_1 \xrightarrow{call\ proc} n_2$, we add the transition rule $\langle n_1, \gamma \rangle \hookrightarrow \langle proc_e, \{n_2\}\gamma \rangle$, for every $proc_e \in \varrho(n_1)(proc)$. This rule moves the program's control point to the entry point of the procedure $proc$, and pushes the return address n_2 onto the stack. Here, we let $proc_e$ be in $\varrho(n_1)(proc)$ because in assembly code, the operand of a call instruction can be any expression including the address of an instruction;
  - if $e = n_1 \xrightarrow{pop\ exp} n_2$, we add the transition rule $\langle n_1, \gamma \rangle \hookrightarrow \langle n_2, \epsilon \rangle$ which moves the program's control point to n_2 and pops the topmost symbol from the stack;
  - if $e = n_1 \xrightarrow{ret} n_2$, we add a transition rule $\langle n_1, \gamma \rangle \hookrightarrow \langle addr, \epsilon \rangle$ for every $addr \in \gamma$. This moves the program's control point to every address $addr$ in γ, and pops the topmost symbol from the stack;
  - if $e = n_1 \xrightarrow{cjmp\ e} n_2$ where $cjmp$ denotes a conditional jump instruction (je, jg, etc.). Let $flag$ be the flag register (ZF,CF, etc.) of $cjmp$. Depending on whether the flag register satisfies the condition of $cjmp$ or not (i.e., whether $false \in \varrho(n_1)(flag)$ or not), we add the transition rules $r_1 = \langle n_1, \gamma \rangle \hookrightarrow \langle n_2, \gamma \rangle$ and/or $r_2 = \langle n_1, \gamma \rangle \hookrightarrow \langle addr, \gamma \rangle$ for every $addr \in \varrho(n_1)(e)$. r_1 moves the program's control point to n_2 whereas r_2 moves the programs's control point to the address $addr$ that corresponds to the value of e at point n_1.
  - if $e = n_1 \xrightarrow{i} n_2$ is any other transition, we add a transition rule $r_1 = \langle n_1, \gamma \rangle \hookrightarrow \langle n_2, \gamma \rangle$ which moves the program's control point from n_1 to n_2 without changing the stack.

Note that in our modeling, the PDS control locations correspond to the program's control points, and the PDS stack mimics the program's execution stack. The above transition rules allow the PDS to mimic the behavior of the program's stack. This is different from standard program translations to PDSs where the control points of the program are stored in the stack [13, 8]. These standard translations assume that the program follows a standard compilation model, where the return addresses are never modified. We do not make such assumptions since behaviors where the return addresses are modified can occur in malicious code. We only make the assumption that pushes and pops can be done only using *push*, *pop*, *call*, and *return* operations, not by manipulating the stack pointer. Our translation is also more precise than the one given in [23]. Indeed, here the stack content is (an over-approximation of) the program's stack, whereas in [23], the stack contains the names of the pushed registers, not their values. For example, in [23], a push instruction of the form $n_1 \xrightarrow{push\ eax} n_2$ is modeled by a push rule where the name of the register *eax* is pushed onto the stack, whereas in this work, we push the possible values of *eax* onto the stack.

3 Malicious Behavior Specification

In this section, we recall the definition of the Stack Computation Tree Predicate Logic (SCTPL) [23], and show how we can use it to specify malicious behaviors in a more precise and succinct way than done in [23].

3.1 Environments, predicates and regular variable expressions

Hereafter, we fix the following notations. Let $\mathcal{X} = \{x_1, x_2, ...\}$ be a finite set of variables ranging over a finite domain $\mathcal{D}$. Let $B : \mathcal{X} \cup \mathcal{D} \longrightarrow \mathcal{D}$ be an environment function that assigns a value $c \in \mathcal{D}$ to each variable $x \in \mathcal{X}$ and such that $B(c) = c$ for every $c \in \mathcal{D}$. $B[x \leftarrow c]$ denotes the environment function such that $B[x \leftarrow c](x) = c$ and $B[x \leftarrow c](y) = B(y)$ for every $y \neq x$. Let $\mathcal{B}$ be the set of all the environment functions.

Let $AP = \{a, b, c, ...\}$ be a finite set of atomic propositions, $AP_{\mathcal{X}}$ be a finite set of atomic predicates in the form of $b(\alpha_1, ..., \alpha_m)$ such that $b \in AP$, $\alpha_i \in \mathcal{X} \cup \mathcal{D}$ for every $1 \leq i \leq m$, and $AP_{\mathcal{D}}$ be a finite set of atomic predicates of the form $b(\alpha_1, ..., \alpha_m)$ such that $b \in AP$, $\alpha_i \in \mathcal{D}$ for every $1 \leq i \leq m$.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, let $\mathcal{R}$ be a finite set of regular variable expressions over $\mathcal{X} \cup \Gamma$ given by: $e ::= \emptyset \mid \epsilon \mid a \in \mathcal{X} \cup \Gamma \mid e + e \mid e \cdot e \mid e^*$.

The language $L(e)$ of a regular variable expression e is a subset of $P \times \Gamma^* \times \mathcal{B}$ defined inductively as follows: $L(\emptyset) = \emptyset$; $L(\epsilon) = \{(\langle p, \epsilon\rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(x)$, where $x \in \mathcal{X}$ is the set $\{(\langle p, \gamma\rangle, B) \mid p \in P, \gamma \in \Gamma, B \in \mathcal{B} : B(x) = \gamma\}$; $L(\gamma)$, where $\gamma \in \Gamma$ is the set $\{(\langle p, \gamma\rangle, B) \mid p \in P, B \in \mathcal{B}\}$; $L(e_1 + e_2) = L(e_1) \cup L(e_2)$; $L(e_1 \cdot e_2) = \{(\langle p, \omega_1\omega_2\rangle, B) \mid (\langle p, \omega_1\rangle, B) \in L(e_1);\ (\langle p, \omega_2\rangle, B) \in L(e_2)\}$; and $L(e^*) = \{(\langle p, \omega\rangle, B) \mid B \in \mathcal{B}$ and $\omega = \omega_1 \cdots \omega_n$, s.t. $\forall i, 1 \leq i \leq n, (\langle p, \omega_i\rangle, B) \in L(e)\}$. E.g., $(\langle p, \gamma_1\gamma_1\gamma_2\rangle, B)$ is an element of $L(x^*\gamma_2)$ when $B(x) = \gamma_1$.

3.2 Stack Computation Tree Predicate Logic

A SCTPL formula is a CTL formula where predicates and regular variable expressions are used as atomic propositions, and where quantifiers over variables are used. Using regular

variable expressions allows to express predicates on the stack content of the PDS. For technical reasons, we suppose w.l.o.g. that formulas are given in positive normal form, i.e., negations are applied only to atomic propositions. More precisely, the set of *SCTPL formulas* is given by (where $x \in \mathcal{X}$, $a(x_1, ..., x_n) \in AP_{\mathcal{X}}$ and $e \in \mathcal{R}$):

$$\begin{aligned}\varphi ::= a(x_1, ..., x_n) \mid \neg a(x_1, ..., x_n) \mid e \mid \neg e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x\, \varphi \\ \mid \exists x\, \varphi \mid AX\varphi \mid EX\varphi \mid A[\varphi U \varphi] \mid E[\varphi U \varphi] \mid A[\varphi R \varphi] \mid E[\varphi R \varphi]\end{aligned}$$

Let φ be a SCTPL formula. The closure $cl(\varphi)$ denotes the set of all the subformulas of φ including φ.

Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ s.t. $\Gamma \subseteq \mathcal{D}$, Let $\lambda : AP_{\mathcal{D}} \rightarrow 2^P$ be a labeling function that assigns a set of control locations to a predicate. Let $c = \langle p, w \rangle$ be a configuration of $\mathcal{P}$. $\mathcal{P}$ satisfies a SCTPL formula ψ in c, denoted by $c \models_\lambda \psi$, iff there exists an environment $B \in \mathcal{B}$ s.t. $c \models^B_\lambda \psi$, where $c \models^B_\lambda \psi$ is defined by induction as follows:

- $c \models^B_\lambda a(x_1, ..., x_n)$ iff $p \in \lambda\Big(a(B(x_1), ..., B(x_n))\Big)$.
- $c \models^B_\lambda \neg a(x_1, ..., x_n)$ iff $p \notin \lambda\Big(a(B(x_1), ..., B(x_n))\Big)$.
- $c \models^B_\lambda e$ iff $(c, B) \in L(e)$.
- $c \models^B_\lambda \neg e$ iff $(c, B) \notin L(e)$.
- $c \models^B_\lambda \psi_1 \wedge \psi_2$ iff $c \models^B_\lambda \psi_1$ and $c \models^B_\lambda \psi_2$.
- $c \models^B_\lambda \psi_1 \vee \psi_2$ iff $c \models^B_\lambda \psi_1$ or $c \models^B_\lambda \psi_2$.
- $c \models^B_\lambda \forall x\, \psi$ iff $\forall v \in \mathcal{D},\ c \models^{B[x \leftarrow v]}_\lambda \psi$.
- $c \models^B_\lambda \exists x\, \psi$ iff $\exists v \in \mathcal{D}$ *s.t.* $c \models^{B[x \leftarrow v]}_\lambda \psi$.
- $c \models^B_\lambda AX\, \psi$ iff $c' \models^B_\lambda \psi$ for every successor c' of c.
- $c \models^B_\lambda EX\, \psi$ iff there exists a successor c' of c s.t. $c' \models^B_\lambda \psi$.
- $c \models^B_\lambda A[\psi_1 U \psi_2]$ iff for every path $\pi = c_0, c_1, ...,$ of $\mathcal{P}$ with $c_0 = c$, $\exists i \geq 0$ s.t. $c_i \models^B_\lambda \psi_2$ and $\forall 0 \leq j < i : c_j \models^B_\lambda \psi_1$.
- $c \models^B_\lambda E[\psi_1 U \psi_2]$ iff there exists a path $\pi = c_0, c_1, ...,$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\exists i \geq 0,\ c_i \models^B_\lambda \psi_2$ and $\forall 0 \leq j < i, c_j \models^B_\lambda \psi_1$.
- $c \models^B_\lambda A[\psi_1 R \psi_2]$ iff for every path $\pi = c_0, c_1, ...,$ of $\mathcal{P}$ with $c_0 = c$, $\forall i \geq 0$ s.t. $c_i \not\models^B_\lambda \psi_2$, $\exists 0 \leq j < i$ s.t. $c_j \models^B_\lambda \psi_1$.
- $c \models^B_\lambda E[\psi_1 R \psi_2]$ iff there exists a path $\pi = c_0, c_1, ...,$ of $\mathcal{P}$ with $c_0 = c$ s.t. $\forall i \geq 0$ s.t. $c_i \not\models^B_\lambda \psi_2$, $\exists 0 \leq j < i$ s.t. $c_j \models^B_\lambda \psi_1$.

Intuitively, $c \models^B_\lambda \psi$ holds iff the configuration c satisfies the formula ψ under the environment B. Note that a path π satisfies $\psi_1 R \psi_2$ iff either ψ_2 holds everywhere in π, or the first occurrence in the path where ψ_2 does not hold must be preceded by a position where ψ_1 holds.

Theorem 1. *[23] Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a SCTPL formula ψ, whether a configuration of $\mathcal{P}$ satisfies ψ can be decided.*

3.3 Using SCTPL formulas in a precise manner

In [23], the stack alphabet Γ (which is a subset of the domain $\mathcal{D}$) we considered consists of the set of registers **R** and the set of the return addresses of the different calls. As explained in the introduction, using the names of the registers instead of their values is not robust and is not very precise. To sidestep this problem, we propose in this work to use the values

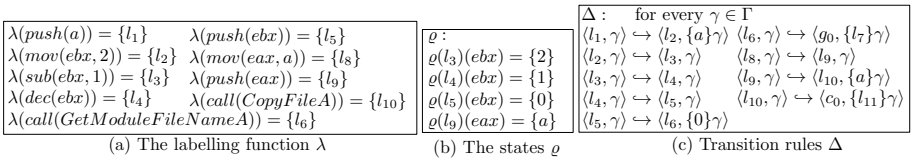

Fig. 2. (a) The labeling function λ, (b) the states ϱ and (c) Transition rules Δ, where g_0 and c_0 are entry points of the function GetModuleFileNameA and CopyFileA, respectively, and l_{11} is the next location of l_{10}

of the registers instead of their names. Hence, in this work, the stack alphabet Γ consists of sets of (over-approximations of) values of registers (elements of $2^{\mathbb{Z}}$), together with the return addresses of the calls.

An illustrating example. Let us consider the fragment of Figure 1(d) and the SCTPL formula Ω described in the introduction. Suppose the oracle O provides the function ρ of Figure[2] 2(b). Then, we have:

- $\Gamma = \{ \{a\}, \{0\}, \{l_7\}, \{l_{11}\} \}$ is the stack alphabet, where l_{11} is the location after l_{10};
- $\mathcal{R} = \{\{0\}\, m\, \Gamma^*, m\Gamma^*\}$ is the set of regular variable expressions in the formula Ω;
- $AP = \{call, mov, sub, dec, push\}$ is the set of atomic propositions corresponding to the instructions of the program;
- $AP_{\mathcal{D}} = \{push(a), mov(ebx, 2), sub(ebx, 1), dec(ebx), push(ebx), mov(eax, a), push(eax), call(GetModuleFileNameA), call(CopyFileA)\}$ is the set of predicates that appear in the program;
- $\mathcal{D} = \{\{a\}, \{0\}, \{l_7\}, \{l_{11}\}, 1, 2, a, eax, ebx, GetModuleFileNameA, CopyFileA\}$;
- The labeling function λ is shown Figure 2(a).
- The set of transition rules Δ of the PDS modeling this fragment is shown in Figure 2(c), where g_0 is the entry point of the procedure GetModuleFileNameA and c_0 is the entry point of the procedure CopyFileA.

3.4 Specifying Malicious Behaviors in SCTPL

We show in this section how SCTPL allows to precisely and succinctly specify several malware behaviors.

Data-stealing Malware. The main purpose of a data-stealing malware is to steal the user's personal confidential data such as username, password, credit card number, etc and send it to another computer (usually the malware writer). The typical behavior of data-stealing malware can be summarized as follows: the malware will first call the API function *ReadFile* in order to load some file of the victim into memory. To do this, it needs to call this function with a file pointer f (i.e., the return value of the calling function *OpenFile*) as the first parameter and a buffer m as the second parameter (m corresponds to the address of a memory location), i.e., with f m on the top of the stack since in assembly, function parameters are passed through the

```
push m
push f
call ReadFile
...
push m
push c
call send
```

Fig. 3: Data-stealing malware.

[2] we give only the values of ρ that are needed to compute the transition relation Δ of Figure 2(c)

stack. Then, the malware will send its file (whose data is pointed by m) to another computer using the function *send*. It needs to call *send* with a connection c (i.e., the return value of the calling function *socket*) as first parameter and the buffer m as the second parameter, i.e., with $c\ m$ on the top of the stack. Figure 3 shows a disassembled fragment of a malware corresponding to this typical behavior. Before calling the function *ReadFile*, it pushes the two parameters m and f onto the stack. Later it calls the function *send* after pushing the two parameters m and c onto the stack. (since in assembly, function parameters are passed through the stack.) This behavior can be expressed by the following SCTPL formula Ω_{ds}.

$$\Omega_{ds} = \exists m\ \ \mathbf{EF}\Big(call(ReadFile) \wedge \Gamma\ m\ \Gamma^* \wedge \mathbf{AF}(call(send) \wedge \Gamma\ m\ \Gamma^*)\Big)$$

where the regular variable expression $\Gamma\ m\ \Gamma^*$ states that the second value of the stack is m (corresponding to the second parameter of the function *ReadFile* and *send*). Ω_{ds} states that there exists an address m which is the second parameter when calling *ReadFile*, and such that later, eventually, *send* will be called with m as its second parameter.

Kernel32.dll base address viruses. Many of Windows viruses use an API to achieve their malicious tasks. The Kernel32.dll file includes several API functions that can be used by the viruses. In order to use these functions, the viruses have to find the entry addresses of these API functions. To do this, they need to determine the Kernel32.dll entry point. They determine first the Kernel32.dll PE header in memory and use this information to locate Kernel32.dll export section and find the entry addresses of the API functions. For this, the virus looks first for the DOS header (the first word of the DOS header is *5A4Dh* in hex (*MZ* in ascii)); and then looks for the PE header (the first two words of the PE header is *4550h* in hex (*PE*00 in ascii)). Figure 4 presents a disassembled code fragment performing this malicious behavior. This can be specified in SCTPL as follows:

$l_1 : cmp\ [eax], 5A4Dh$
$jnz\ l_2$
$\cdots$
$cmp\ [ebx], 4550h$
$jz\ l_3$
$l_2 :\ \ldots$
$jmp\ l_1$
l_3

Fig. 4: Virus.

$$\psi_{wv} = EG\Big(EF(\exists r_1\ cmp(r_1, 5A4Dh) \wedge EF\ \exists r_2\ cmp(r_2, 4550h))\Big).$$

This SCTPL formula expresses that the program has a loop such that there are two variables r_1 and r_2 such that first, r_1 is compared to *5A4Dh*, and then r_2 is compared to *4550h*. Note that this formula can detect all the class of viruses that have such behavior.

Obfuscated calls. Virus writers try to obfuscate their code by e.g. hiding the calls to the operating system. For example, a *call* instruction can be replaced by pushes and jumps. Figure 5 shows two equivalent fragments achieving a "call" instruction. Figure 5(a) shows a normal call/ret where the function f consists just of a *return* instruction. When control point f is reached, the *return* instruction moves the control point to l_1 which is the return address of the call instruction (at l_0). As shown in Figure 5(b), the *call* can be equivalently substituted by two other instructions, where *push* l'_2 pushes the return address l'_2 onto the stack, and *jmp* f moves the control point to the entry point of f. These instructions do exactly the same thing than the *call* instruction. When reaching the control point f, the *ret* instruction will pop the stack and

l_0 : call f	l'_0 : push l'_2
l_1 : ...	l'_1 : jmp f
	l'_2 : ...
f: ret	
	f: ret
(a)	(b)

Fig. 5: (a) Normal call. (b) Obfusated call

thus, move the control point to l'_2. Such obfuscated calls can be described by the following SCTPL formula:

$$\psi_{oc} = \exists\, addr\ \mathbf{E}[\neg(\exists\, proc\ call(proc) \wedge\ \mathbf{AX}\ addr\Gamma^*)\ \mathbf{U}\ (ret \wedge addr\Gamma^*)]$$

The subformula $(\exists\, proc\ call(proc) \wedge \mathbf{AX}\ addr\Gamma^*)$ means that there exists a procedure call having *addr* as return address, since when a procedure call is made, the program will push its corresponding return address *addr* to the stack, and thus, at the next step, we will have *addr* on the top of the stack (i.e., $addr\Gamma^*$). The subformula $(ret \wedge addr\Gamma^*)$ expresses that we have a return instruction with *addr* on the top of the stack, i.e., a return instruction that will return to *addr*. Thus the formula ψ_{oc} expresses that there exists a return address *addr* such that there exists a path where there is no call to a procedure *proc* having *addr* as return address until a return instruction with *addr* as return address occurs. This formula can then detect a return that does not correspond to a call.

Obfuscated returns. Virus writers usually obfuscate the returns of their calls in order to make it difficult to manually or automatically analyze their code. Benign programs move the control point to the return address using the *ret* instruction. Viruses may replace the *ret* instruction by other equivalent instructions such as *pop eax*, *jmp eax*, etc. E.g., the program in Figure 6 is a disassembled fragment from the virus Klinge that pops the return address *00401028* from the stack. This phenomenon can be detected by the following specification:

```
00401023: call 004011CE
00401028: ...
    ...
004011CE:  ...
    ...
0040121A: pop eax
```

Fig. 6: Fragment of the Virus Klinge

$$\psi_{or} = \mathbf{AG}\left(\forall proc \forall addr((call(proc) \wedge \mathbf{AX}\ addr\Gamma^*) \Longrightarrow \mathbf{AF}(ret \wedge addr\Gamma^*))\right).$$

ψ_{or} expresses that for every procedure *proc*, if *proc* is called with *addr* as the return address of the caller, then there exists a *ret* instruction which will return to *addr*. Indeed, since when an assembly program runs, if an instruction *call proc* is executed, then the return address *addr* of the caller is pushed onto the stack. Thus, in the subformula $call(proc) \wedge \mathbf{AX}\ addr\Gamma^*$, *addr* refers to the return address of the call, because this subformula expresses that in all the immediate successors of the call, *addr* is on the top of the stack. Moreover, $ret \wedge addr\Gamma^*$ means that when the return is executed, then the return address *addr* should be on the top of the stack.[3]

Appending viruses. An appending virus is a virus that inserts a copy of its malicious code at the end of the target file. To do this, the virus has to first calculate its real absolute address in the memory, because the real OFFSET of the virus' variables depends on the size of the infected file. To achieve this, the viruses have to call the routine in Figure 7 (this code is a fragment of the virus Alcaul.b). The instruction *call* l_2 will push the return address l_2 onto the stack. Then, the *pop* instruction will put the value of this address into the register *eax*. In this way, the virus can get its real absolute address in the memory. This malicious behavior can be detected using the specification ψ_{or}, since there does not exist any *return* instruction corresponding to the *call* instruction.

```
l1 : call l2
l2 : pop eax
    ...
```

Fig. 7:

[3] Note that for the case of a procedure that has a possibly infinite loop, this specification can detect a suspected malware. This formula can be changed slightly to avoid this. We do not present this here for the sake of presentation.

4 SCTPL\X and the Collapsing Abstraction

As discussed in [23], the algorithm underlying Theorem 1 is very expensive. It is exponential on the size of the PDS. Thus, it is important to model binary programs by PDSs with small sizes. For this, we propose in this section to use the *collapsing abstraction* to drastically reduce the size of the PDS model of the program. Moreover, we consider SCTPL\X, a fragment of SCTPL that uses the next time operator X only to specify the return addresses of the callers. All the malicious behaviors that we considered can be specified using SCTPL\X formulas. We show that the collapsing abstraction preserves SCTPL\X formulas.

4.1 SCTPL\X

SCTPL\X is defined by the following, where $a(x_1, ..., x_n) \in AP_{\mathcal{X}}$, *func* is a function, $e \in \mathcal{R}$ and $r \in \Gamma \cup \mathcal{X}$:

$$\varphi ::= a(x_1, ..., x_n) \mid \neg a(x_1, ..., x_n) \mid e \mid \neg e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x\, \varphi \mid \exists x\, \varphi$$
$$\mid call(func) \wedge AX\, r\Gamma^* \mid A[\varphi U \varphi] \mid E[\varphi U \varphi] \mid A[\varphi R \varphi] \mid E[\varphi R \varphi]$$

Intuitively, SCTPL\X is the sub-logic of SCTPL where the next time operator **X** is used only to specify the return addresses of the callers. Indeed, the SCTPL formula $call(func) \wedge \mathbf{AX}\ r\Gamma^*$ means that r is the return address of the function $func$, since the return address is always pushed onto the stack when a function is called. The subformula $\mathbf{AX}\ r\Gamma^*$ ensures that the return address r is the topmost symbol of the stack at the next control point after calling the function $func$.

SCTPL\X is sufficient to specify malware. Indeed, arbitrary SCTPL formulas of the form $AX\psi$ or $EX\psi$ that cannot be expressed by SCTPL\X should not be used for malware specifications since such formulas are not robust. Indeed, suppose that at some control point n, a piece of malware satisfies a formula $AX\psi$. Then inserting some dead code at control point n will make the formula $AX\psi$ unsatisfiable. Thus, if a specification that involves such formulas can detect a given malware, it cannot detect variants of this malware where dead code is added at some locations. It is then not recommended to use such subformulas for malware specification. Thus, to make these specifications of malicious behaviors more robust, we should specify these behaviors by $AF\psi$ or $EF\psi$.

4.2 The collapsing abstraction

Given a program, the collapsing abstraction reduces the size of the program model by removing all the irrelevant instructions of the program, i.e., all the instructions that do not change the stack (instructions using push or pop are not removed), nor the control flow of the program (instructions using jump-like operators, e.g., jmp, jz, etc. are not removed); as well as the instructions whose operators do not appear in the considered SCTPL formula.

More precisely, consider a SCTPL formula ψ and a binary program. Let $G = (N, I, E)$ and ϱ be respectively the CFG and the state function provided by the oracle O. Let $op(b(a_1, ..., a_n))$ denote the operator b for every instruction $b(a_1, ..., a_n) \in I$. Let $I_\psi = \{b \mid \exists b(x_1, ..., x_n) \in cl(\psi)\}$ be the set of operators that appear in the formula ψ, $I_{stack} = \{push, pop, call, ret\}$ be the set of operators that modify the program's stack, and $I_{jump} = \{jmp, jz, je, jnz, jne, js, jns, jo, jno, jp, jnp, jpe, jpo, jc, jb, jnae, jnc, jnb, jae, jbe,$

jna, *jnbe*, *ja*, *jl*, *jnge*, *jnl*, *jge*, *jle*, *jng*, *jnle*, *jg*, *jcxz*} be the set of all the jump instructions. Let $N_{target} = \{n \in N \mid \exists n_1 \xrightarrow{b(e)} n_2 \in E \; s.t. \; e \in EXP, n \in \varrho(n_1)(e) \wedge b \in I_{jump} \cup \{call\}\}$ be the set of nodes that can be reached by a *call* or a *jump* instruction of the program.
The collapsing abstraction removes from the program all the instructions whose operators are not in $I_\psi \cup I_{stack} \cup I_{jump}$ and whose control points are not in N_{target}. More precisely, we compute a new control flow graph $G_\psi = (N', I', E')$ such that N' is a subset of N, $I' = \{\bot\} \cup \{i \in I \mid op(i) \in I_\psi \cup I_{stack} \cup I_{jump}\}$, E' is defined as follows: $n \xrightarrow{i} n' \in E'$ iff

- $n \xrightarrow{i} n' \in E$ and $i \in I'$;
- or $i = \bot$ is a fake instruction that we add, $n \in N_{target}$, and $\exists n \xrightarrow{i'} n' \in E$ s.t. $i' \notin I'$;
- or $i = \bot$, there exists in G a path of the form $p \xrightarrow{l_1} n \xrightarrow{i_1} n_1 \xrightarrow{i_2} n_2 \cdots n_{k-1} \xrightarrow{i_k} n' \xrightarrow{l_2} p'$ s.t. $p \xrightarrow{l_1} n \in E'$ and $n' \xrightarrow{l_2} p' \in E'$ are two edges in E' meaning that either l_1 and/or l_2 cannot be removed or is $\bot$, whereas for every $1 \leq j \leq k$, the instruction i_j is removed, i.e., the operator $op(i_j)$ of the instruction i_j is not in $I_\psi \cup I_{stack} \cup I_{jump}$ and for every $1 \leq j \leq k-1$ node n_j is not in N_{target}.

We add the instructions $\bot$ to relate two nodes that are related by a path in G and such that removing the irrelevant instructions could make these nodes disconnected in G'. Note that we do not remove nodes in N_{target} because they could be reached by different paths.

The control flow graph G_ψ can be computed in linear time:

Lemma 1. *Given a SCTPL formula ψ, and a control flow graph G, G_ψ can be effectively computed in linear time.*

We can show that this abstraction preserves formulas that do not involve properties about the next state. Formulas using the **X** operator in an arbitrary manner are not preserved since this abstraction removes instructions from the program. However, formulas of the form $call(func) \wedge AX \, r\Gamma^*$ are preserved since they express that a call to the function $func$ is made, and r is the return address of this call. Therefore, such a formula is related to the *single* instruction $call(func)$. So, removing the irrelevant instructions as described above will not change the satisfiability of this formula. Thus, we can show that this abstraction preserves SCTPL\X formulas:

Theorem 2. *Let ψ be a SCTPL\X formula. Let $\mathcal{P}$ be the PDS modeling a CFG G w.r.t. a state function ϱ, and let $\mathcal{P}'$ be the PDS modeling the CFG G_ψ w.r.t. the state function ϱ. Then $\mathcal{P}$ satisfies ψ iff $\mathcal{P}'$ satisfies ψ.*

5 Experiments

We implemented our techniques in a tool for malware detection. Our tool gets a binary program as input, and outputs Yes or No, depending on whether the code contains a malicious behavior or not. To implement an oracle $\mathcal{O}$, we use Jakstab [19] and IDA Pro [6]. Jakstab performs static analysis of the binary program and provides a control flow graph and a state function ϱ. However, it does not allow to extract API functions' information and some indirect calls to the API functions. We use IDA Pro to get these informations. We use BDDs to represent sets of environments. To perform SCTPL model-checking of

PDSs, we implement the algorithms of [23]. All the experiments were run on a Linux platform (Fedora 13) with a 2.4GHz CPU, 2GB of memory. The time limit is fixed to 30 minutes.

We evaluated our tool on 200 malwares taken from VX Heavens [15] and 8 benign programs taken from system32 of Microsoft Windows XP: cmd.exe, find.exe, java.exe, notepad.exe, ping.exe, print.exe, regedt.exe and shutdown.exe. Our tool was able to detect all the 200 malwares. Moreover, it reported that the benign programs that we considered are not malicious, except for java.exe. Our tool detected a malicious behavior in this program. This behavior was introduced by the over-approximation provided by Jakstab [19]. The time and memory consumptions are shown in Figures 8 and 9. These figures show the gain in time and memory consumption when the collapsing abstraction is used. The analysis of several examples (such as Bagle.m, print.exe and notepad.exe e.g.) terminates when using the collapsing abstraction, whereas it runs out of memory without this abstraction.

Generator	No. of Variants	Our techniques detection rate	Avira detection rate	kaspersky detection rate	Avast antivirus detection rate	Qihoo 360 detection rate
NGVCK	100	**100**%	0%	23%	18%	68%
VCL32	100	**100**%	0%	2%	100%	99%

Table 1. Detection of variants generated by NGVCK and VCL32.

Furthermore, to compare our techniques with the well-known existing anti-virus tools, and show the robustness of our tool, we automatically generated 200 new malwares using the generators NGVCK and VCL32 available at VX Heavens [15]. We generated 100 malwares using NGVCK, and 100 using VCL32. [24] showed that these systems are the best malware generators, compared to the other generators of VX Heavens [15]. These programs use very sophisticated features such as anti-disassembly, anti-debugging, anti-emulation, and anti-behavior blocking and come equipped with code morphing ability which allows them to produce different-looking viruses. Our results are reported in Table 1. Our techniques were able to detect all these 200 malwares, whereas the four well known and widely used anti-viruses Avira [3], Avast [2], Kaspersky [1] and Qihoo 360 [4] were not able to detect several of these viruses.

References

1. 30 days free kaspersky antivirus. `http://www.kaspersky.com`. Version 12.0.0.374.
2. Free avast antivirus. `http://www.avast.com`. Version 6.0.1367.
3. Free avira antivirus. `http://www.avira.com`. Version 12.0.0.849.
4. Qihoo 360 antivirus. `http://www.360.cn`.
5. 10 most destructive computer worms and viruses ever. `http://wildammo.com/2010/10/12/10-most-destructive-computer-worms-and-viruses-ever/#`, 2010.
6. Disassembler. Technical report, `http://www.hex-rays.com/idapro/`, 2011.
7. D. Babic, D. Reynaud, and D. Song. Malware analysis with tree automata inference. In *CAV*, pages 116–131, 2011.

8. G. Balakrishnan, T. W. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. H. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with codesurfer/x86 and wpds++. In *CAV*, pages 158–163, 2005.
9. J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *SREIS*, 2001.
10. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, 5:263–270, 2009.
11. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, pages 169–186, 2003.
12. M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46, 2005.
13. J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *CAV'01*, LNCS, pages 324–336, 2001.
14. A. Gostev. Kaspersky security bulletin, malware evolution 2010. `http://www.securelist.com/en/analysis/204792161/Kaspersky_Security_Bulletin_Malware_Evolution_2010`, 2011. Kaspersky Lab ZAO.
15. V. Heavens. http://vx.netlux.org.
16. A. Holzer, J. Kinder, and H. Veith. Using verification technology to specify and detect malware. In *EUROCAST*, pages 497–504, 2007.
17. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *DIMVA*, pages 174–187, 2005.
18. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive detection of computer worms using model checking. *IEEE Trans. on Dependable and Secure Computing*, 7(4), 2010.
19. J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *CAV*, pages 423–427, 2008.
20. A. Lakhotia, D. R. Boccardo, A. Singh, and A. Manacero. Context-sensitive analysis of obfuscated x86 executables. In *PEPM*, 2010.
21. A. Lakhotia, E. U. Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Trans. Software Eng.*, 31(11), 2005.
22. P. K. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *IAW*, pages 298–300, 2003.
23. F. Song and T. Touili. Pushdown model checking for malware detection. In *TACAS*, pages 110–125, 2012.
24. W. Wong. Analysis and detection of metamorphic computer viruses. Master's thesis, San Jose State University, 2006.

Algorithm 1: The Algorithm computing G_ψ.

Input : A SCTPL formula ψ, and a CFG $G = (N, I, E)$ with its state function ϱ;
Output: The CFG G_ψ;

1 Let $N_{target} \subseteq N$ be the set of nodes that are targets of some instruction of I;
2 Let $I_\psi = \{b \mid \exists b(x_1, ..., x_n) \in cl(\psi)\}$;
3 Let $I_{jump} = \{jmp, jz, je, jnz, jne, js, jns, jo, jno, jp, jnp, jpe, jpo, jc, jb, jnae, jnc, jnb, jae, jbe, jna, jnbe, ja, jl, jnge, jnl, jge, jle, jng, jnle, jg, jcxz\}$;
4 Let $I_{stack} = \{call, ret, push, pop\}$;
5 Let $G_\psi = (N', I', E')$ such that $N' = E' = \emptyset, I' = \{\bot\}$;
6 **for** $n \xrightarrow{b(\alpha_1,...,\alpha_m)} n' \in E$ **do**
7 **if** $b \in I_\psi \cup I_{jump} \cup I_{stack}$ *or* $n \in N_{target}$ **then**
8 Add n, n' into N';
9 Add $n \xrightarrow{I} n'$ into E', where $I = b \in I_\psi \cup I_{jump} \cup I_{stack}\ ?\ b(\alpha_1, ..., \alpha_m) : \bot$;
10 Let $E'' = E \setminus E'$;
11 **for** $e_1 = n_1 \xrightarrow{i_1} n_2 \in E''$ **do**
12 **if** $\exists e_2 = n_2 \xrightarrow{i_2} n_3 \in E''$ **then**
13 $E'' = E'' \cup \{n_1 \xrightarrow{\perp} n_3\} \setminus \{e_1, e_2\}$;
14 **else**
15 Add $n_1 \xrightarrow{\perp} n_2$ into E';
16 Add n_1 and n_2 into N';

A Appendix

A.1 Proof of Lemma 1

Lemma 1. *Given a SCTPL formula ψ, and a control flow graph $G = (N, I, E)$ with its state function ϱ, G_ψ can be effectively computed in linear time.*

Proof: We proceed by giving an algorithm which exactly computes the control flow graph G_ψ.

Algorithm 1 computes exactly the control flow graph G_ψ. Initially, we can compute all the possible target nodes N_{target} as follows: $n \in N_{target}$ iff $n_1 \xrightarrow{b(e)} n_2 \in E$ such that $n \in \varrho(n_1)(e)$ and $b \in I_{jump}$ or $b = call$. This can be done in time $\mathbf{O}(I)$.

Then, We proceed by two phrases: Lines 6-9 and Lines 11-16. The first phrase (Lines 6-9) computes all the nodes and edges such that either the starting node n of an edge in E is a target (i.e., $n \in N_{target}$), or the operator b of the instruction changes the stack, or changes the control flow, or is used in ψ (i.e., $b \in I_\psi \cup I_{jump} \cup I_{stack}$). This phrase can be done in time $\mathbf{O}(E)$.

E'' is the rest edges of E after the first phrase (line 10). The instruction of every edge in E'' will not change the control flow, the stack or the operator of the instruction is not used in ψ.

The second phrase will remove two edges $n_1 \xrightarrow{i_1} n_2$ and $n_2 \xrightarrow{i_2} n_3$ from E'' and add one edge $n_1 \xrightarrow{\perp} n_3$ into E''. By doing this, each path $n_1 \xrightarrow{i_1} n_2 \xrightarrow{i_2} n_3, ..., \xrightarrow{i_k} n_{k+1}$ such

that there does not exist $n \xrightarrow{i} n_1 \in E''$ or $n_{k+1} \xrightarrow{i} n \in E''$. will be substituted by an edge $n_1 \xrightarrow{\perp} n_{k+1}$ which will be added into E' (line 15). The second phrase can be done in time $\mathbf{O}(E'')$.

W.l.o.g., for each edge added in phrase two, we can assume that there exist two edges $n \xrightarrow{i} n_1 \in E'$ and/or $n_{k+1} \xrightarrow{i} n \in E'$. Indeed, if these edges do not exist, then the node n_1 (resp. n_{k+1}) has no predecessor (resp. successor). We can add a fake instruction $n \xrightarrow{i} n_1 \in E'$ and/or $n_{k+1} \xrightarrow{i} n \in E'$ into E'. Since these nodes are not reachable which does not have any side-effect for Theorem 2.

Thus, G_ψ can be efficiently computed in linear time. □

A.2 Proof of Theorem 2

Theorem 2. *Let ψ be a SCTPL\X formula. Let $\mathcal{P}$ be the PDS modelling a CFG G w.r.t. the state function ϱ, and let $\mathcal{P}'$ be the PDS modeling the CFG G_ψ w.r.t. the state function ϱ. Then $\mathcal{P}$ satisfies ψ iff $\mathcal{P}'$ satisfies ψ.*

Proof: We proceed by proving a more general case where SCTPL\X is given by: (where $a(x_1, ..., x_n) \in AP_\chi$ and $e \in \mathcal{R}$):

$$\varphi ::= a(x_1, ..., x_n) \mid e \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists x\, \varphi \mid EXe \mid EX\neg e \mid EG\varphi \mid E[\varphi U \varphi]$$

In this definition, the next time operator $\mathcal{X}$ can only specify regular variable expressions and their negation. This definition is more general than the definition given in Section 4.1. We show that:

> Let ψ be a SCTPL\X formula given by the above definition. Let $\mathcal{P}$ be the PDS modelling a CFG G w.r.t. the state function ϱ, and let $\mathcal{P}'$ be the PDS modeling the CFG G_ψ w.r.t. the state function ϱ. Then $\mathcal{P}$ satisfies ψ iff $\mathcal{P}'$ satisfies ψ.

Let $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{P}' = (P', \Gamma', \Delta')$.

Since the abstraction does not remove any node where the instruction will change the stack or is a target, or change the control flow. Thus, we obtain that the initial configuration of $\mathcal{P}$ and $\mathcal{P}'$ are identical and $\Gamma = \Gamma'$. Let c_0 be the initial configuration of $\mathcal{P}$ and $\mathcal{P}'$.

First, let us characterize some properties of PDS modeling the CFG w.r.t. ϱ. Due to the construction of the PDS from CFG, PDS do not have any rewrite transition rule, i.e., $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \notin \Delta$ if $\gamma \neq \gamma'$.

Since the abstraction only remove edges $n \xrightarrow{b(a_1,...,a_m)} n'$ when $b \notin I_\psi \cup I_{stack} \cup I_{jump}$ and $n \notin N_{target}$, we can obtain that

- $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma_1\gamma_2\rangle \in \Delta$ iff $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma_1\gamma_2\rangle \in \Delta'$;
- $\langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle \in \Delta$ iff $\langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle \in \Delta'$;
- $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta'$ iff
  - $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ and $p \in \varrho(n')(e)$ for some $n' \in N$, i.e., p is the target; or
  - $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta$ and the operator of the instruction at p is in $I_\psi \cup I_{jump}$; or
  - $\gamma = \gamma'$ and, there exist $p_1, p_2, ..., p_k \in P$ such that $\langle p, \gamma\rangle \rightsquigarrow_{\mathcal{P}} \langle p_1, \gamma\rangle \rightsquigarrow_{\mathcal{P}} \langle p_2, \gamma\rangle, ..., \rightsquigarrow_{\mathcal{P}} \langle p_k, \gamma\rangle \rightsquigarrow_{\mathcal{P}} \langle p', \gamma'\rangle$, $p_1, ..., p_k \notin N_{target}$ and the operator b of the instruction at $p, p', p_1, ..., p_k$ is not in $I_{jump} \cup I_\psi \cup I_{stack}$.

Now, let us prove that the PDS $\mathcal{P}$ satisfies ψ iff the PDS $\mathcal{P}'$ satisfies ψ. It is sufficient to show that the initial configuration c_0 of the PDS $\mathcal{P}$ satisfies ψ iff the initial configuration c_0 of the PDS $\mathcal{P}'$ satisfies ψ. For simplify the representation, a configuration c of $\mathcal{P}$ (resp. $\mathcal{P}'$) satisfying ψ under the environment B is denoted by $c \models^B_\lambda \psi$ (resp. $c \vdash^B_\lambda \psi$, $\vdash$ has the same semantics as for $\models$). Since each instruction $i \in I$ removed by the abstraction does not contain any operator that used by the SCTPL\X formula ψ, i.e, there does not exist any atomic predicate $b(x_1, ..., x_n)$ in ψ such that b is a operator of i. Let λ and λ' be the labelling function of the PDS $\mathcal{P}$ and $\mathcal{P}'$ respectively. Since the CFG G and G_ψ have the same state function ϱ, we obtain that for every $n \in P'$ and $b(a_1, ..., a_n) \in I'$: $n \in \lambda(b(a_1, ..., a_n))$ iff $n \in \lambda'(b(a_1, ..., a_n))$.

Since the initial configuration c_0 is a configuration of $\mathcal{P}$ as well as $\mathcal{P}'$ and every configuration c of $\mathcal{P}'$ is also a configuration of $\mathcal{P}$ due to $P' \subseteq P$ and $\Gamma' = \Gamma$.

It is sufficient to prove that for every configuration $\langle n, \omega\rangle \in P' \times \Gamma^*$, $\langle n, \omega\rangle \models^B_\lambda \psi$ iff $\langle n, \omega\rangle \vdash^B_{\lambda'} \psi$.

We proceed by applying the induction on the structure of the SCTPL\X formula ψ.

- **Case** $\psi = b(x_1, ..., x_n)$: since $\langle n, \omega\rangle \models^B_\lambda b(x_1, ..., x_n)$ iff $n \in \lambda(b(B(x_1), ..., B(x_n)))$, $\langle n, \omega\rangle \vdash^B_{\lambda'} b(x_1, ..., x_n)$ iff $n \in \lambda'(b(B(x_1), ..., B(x_n)))$. Since for every $n \in P'$ and $b(a_1, ..., a_n) \in I'$: $n \in \lambda(b(a_1, ..., a_n))$ iff $n \in \lambda'(b(a_1, ..., a_n))$. We obtain that $\langle n, \omega\rangle \models^B_\lambda b(x_1, ..., x_n)$ iff $c \vdash^B_{\lambda'} b(x_1, ..., x_n)$.
- **Case** $\psi = e \in \mathcal{R}$: $\langle n, \omega\rangle \models^B_\lambda e$ iff $(\langle n, \omega\rangle, B) \in L(e)$ iff $c \vdash^B_{\lambda'} e$.
- **Case** $\psi = \exists x\psi'$: $\langle n, \omega\rangle \models^B_\lambda \exists x\psi'$ iff there exists a value $v \in \mathcal{D}$ such that $\langle n, \omega\rangle \models^{B[x\leftarrow v]}_\lambda \psi'$. $\langle n, \omega\rangle \vdash^B_{\lambda'} \exists x\psi'$ iff there exists a value $v \in \mathcal{D}$ such that $\langle n, \omega\rangle \vdash^{B[x\leftarrow v]}_{\lambda'} \psi'$.

  By applying the induction hypothesis, $\langle n, \omega\rangle \models^{B[x\leftarrow v]}_\lambda \psi'$ iff $\langle n, \omega\rangle \vdash^{B[x\leftarrow v]}_{\lambda'} \psi'$. Thus, $\langle n, \omega\rangle \models^B_\lambda \exists x\psi'$ iff $\langle n, \omega\rangle \vdash^B_{\lambda'} \exists x\psi'$.
- **Case** $\psi = \neg\psi'$: $\langle n, \omega\rangle \models^B_\lambda \neg\psi'$ iff $\langle n, \omega\rangle \not\models^B_\lambda \psi'$.
  By applying the induction hypothesis, $\langle n, \omega\rangle \not\models^B_\lambda \psi'$ iff $\langle n, \omega\rangle \nvdash^B_{\lambda'} \psi'$. Thus, $\langle n, \omega\rangle \models^B_\lambda \neg\psi'$ iff $\langle n, \omega\rangle \vdash^B_{\lambda'} \neg\psi'$.
- **Case** $\psi = \psi_1 \wedge \psi_2$: $\langle n, \omega\rangle \models^B_\lambda \psi_1 \wedge \psi_2$ iff $\langle n, \omega\rangle \models^B_\lambda \psi_1$ and $\langle n, \omega\rangle \models^B_\lambda \psi_2$. By applying the induction hypothesis, $\langle n, \omega\rangle \models^B_\lambda \psi_1$ iff $\langle n, \omega\rangle \vdash^B_{\lambda'} \psi_1$ and $\langle n, \omega\rangle \models^B_\lambda \psi_2$ iff $\langle n, \omega\rangle \vdash^B_{\lambda'} \psi_2$.

  Since $\langle n, \omega\rangle \vdash^B_{\lambda'} \psi_1 \wedge \psi_2$ iff $\langle n, \omega\rangle \vdash^B_{\lambda'} \psi_1$ and $\langle n, \omega\rangle \vdash^B_{\lambda'} \psi_2$. We obtain that $\langle n, \omega\rangle \models^B_\lambda \psi_1 \wedge \psi_2$ iff $\langle n, \omega\rangle \vdash^B_{\lambda'} \psi_1 \wedge \psi_2$.
- **Case** $\psi = EXe$: Since $\langle n, \omega\rangle \models^B_\lambda EXe$ there exists a configuration $\langle n', \omega'\rangle$ such that $\langle n, \omega\rangle \rightsquigarrow_\mathcal{P} \langle n', \omega'\rangle$ and $\langle n', \omega'\rangle \models^B_\lambda e$.

  Since $\langle n, \omega\rangle \rightsquigarrow_\mathcal{P} \langle n', \omega'\rangle$, according to definition of the abstraction, we obtain that either $\langle n, \omega\rangle \rightsquigarrow_{\mathcal{P}'} \langle n', \omega'\rangle$, or $\langle n, \omega\rangle \rightsquigarrow_{\mathcal{P}'} \langle n_k, \omega'\rangle$ such that $\langle n, \omega\rangle \rightsquigarrow_\mathcal{P} \langle n', \omega'\rangle \rightsquigarrow_\mathcal{P} \langle n_1, \omega'\rangle \rightsquigarrow_\mathcal{P} \langle n_2, \omega'\rangle, ..., \rightsquigarrow_\mathcal{P} \langle n_{k-1}, \omega'\rangle \rightsquigarrow_\mathcal{P} \langle n_k, \omega'\rangle$.

  Since $\langle n', \omega'\rangle \models^B_\lambda e$ iff $\langle n_k, \omega'\rangle \vdash^B_{\lambda'} e$, we obtain that $\langle n, \omega\rangle \models^B_\lambda EXe$ iff $\langle n, \omega\rangle \vdash^B_{\lambda'} EXe$.
- **Case** $\psi = EX\neg e$ is similar as the case $\psi = EXe$.

– **Case** $\psi = EG\psi'$: Since $\langle n, \omega\rangle \vdash^B_{\lambda'} EG\psi'$ iff $\mathcal{P}'$ has an infinite path $\langle n_0, \omega_0\rangle \rightsquigarrow_{\mathcal{P}'} \langle n_1, \omega_1\rangle \rightsquigarrow_{\mathcal{P}'} \langle n_2, \omega_2\rangle \rightsquigarrow_{\mathcal{P}'} ...$ such that $\langle n_0, \omega_0\rangle = \langle n, \omega\rangle$ and $\langle n_j, \omega_j\rangle \vdash^B_{\lambda'} \psi'$ for every $j \geq 0$.

  According to the definition of abstraction, for every $j \geq 0$: $\langle n_j, \omega_j\rangle \rightsquigarrow_{\mathcal{P}'} \langle n_{j+1}, \omega_{j+1}\rangle$ iff either $\langle n_j, \omega_j\rangle \rightsquigarrow_{\mathcal{P}} \langle n_{j+1}, \omega_{j+1}\rangle$, or $\langle n_j, \omega_j\rangle \rightsquigarrow_{\mathcal{P}} \langle n^j_{k_1}, \omega_j\rangle \rightsquigarrow_{\mathcal{P}} \langle n^j_{k_2}, \omega_j\rangle, ... \rightsquigarrow_{\mathcal{P}} \langle n^j_{k_j}, \omega_j\rangle \rightsquigarrow_{\mathcal{P}} \langle n_{j+1}, \omega_{j+1}\rangle$ and the operator b of the instruction at $n_j, n^j_{k_1}, ..., n^j_{k_j}, n_{j+1}$ is not in $I_{jump} \cup I_\psi \cup I_{stack}$. Thus $\langle n^j_{k_m}, \omega_j\rangle \models^B_\lambda \psi'$ iff $\langle n_j, \omega_j\rangle \models^B_\lambda \psi'$ for every $1 \leq m \leq k_j$.

  We obtain that $\langle n, \omega\rangle \vdash^B_{\lambda'} EG\psi'$ iff $\langle n, \omega\rangle \models^B_\lambda EG\psi'$.

– **Case** $\psi = E[\psi_1 U \psi_2]$: Since

  $\langle n, \omega\rangle \models^B_\lambda E[\psi_1 U \psi_2]$ iff there exists $k \geq 0$ such that $\langle n_0, \omega_0\rangle \rightsquigarrow_{\mathcal{P}} \langle n_1, \omega_1\rangle \rightsquigarrow_{\mathcal{P}} \langle n_2, \omega_2\rangle ...\langle n_{k-1}, \omega_{k-1}\rangle \rightsquigarrow_{\mathcal{P}} \langle n_k, \omega_k\rangle$ is a path of $\mathcal{P}$, $\langle n_k, \omega_k\rangle \models^B_\lambda \psi_2$ and $\langle n_i, \omega_i\rangle \models^B_\lambda \psi_1$ for every $0 \leq i < k$ where $\langle n_0, \omega_0\rangle = \langle n, \omega\rangle$

  $\langle n, \omega\rangle \vdash^B_{\lambda'} E[\psi_1 U \psi_2]$ iff there exists $k' \geq 0$ such that $\langle n'_0, \omega'_0\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_1, \omega'_1\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_2, \omega'_2\rangle ...\langle n'_{k'-1}, \omega'_{k'-1}\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_{k'}, \omega'_{k'}\rangle$ is a path of $\mathcal{P}'$, $\langle n'_{k'}, \omega'_{k'}\rangle \vdash^B_{\lambda'} \psi_2$ and $\langle n'_i, \omega'_i\rangle \vdash^B_{\lambda'} \psi_1$ for every $0 \leq i < k'$ where $\langle n'_0, \omega'_0\rangle = \langle n, \omega\rangle$.

  It is sufficient to prove that there exists $k \geq 0$ such that $\langle n_0, \omega_0\rangle \rightsquigarrow_{\mathcal{P}} \langle n_1, \omega_1\rangle \rightsquigarrow_{\mathcal{P}} \langle n_2, \omega_2\rangle ...\langle n_{k-1}, \omega_{k-1}\rangle \rightsquigarrow_{\mathcal{P}} \langle n_k, \omega_k\rangle$ is a path of $\mathcal{P}$, $\langle n_k, \omega_k\rangle \models^B_\lambda \psi_2$ and $\langle n_i, \omega_i\rangle \models^B_\lambda \psi_1$ for every $0 \leq i < k$ where $\langle n_0, \omega_0\rangle = \langle n, \omega\rangle$ iff there exists $k' \geq 0$ such that $\langle n'_0, \omega'_0\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_1, \omega'_1\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_2, \omega'_2\rangle ...\langle n'_{k'-1}, \omega'_{k'-1}\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_{k'}, \omega'_{k'}\rangle$ is a path of $\mathcal{P}'$, $\langle n'_{k'}, \omega'_{k'}\rangle \vdash^B_{\lambda'} \psi_2$ and $\langle n'_i, \omega'_i\rangle \vdash^B_{\lambda'} \psi_1$ for every $0 \leq i < k'$ where $\langle n'_0, \omega'_0\rangle = \langle n, \omega\rangle$. We proceed by proving two directions as follows.

  - Suppose there exists $k \geq 0$ such that $\langle n_0, \omega_0\rangle \rightsquigarrow_{\mathcal{P}} \langle n_1, \omega_1\rangle \rightsquigarrow_{\mathcal{P}} \langle n_2, \omega_2\rangle ...\langle n_{k-1}, \omega_{k-1}\rangle \rightsquigarrow_{\mathcal{P}} \langle n_k, \omega_k\rangle$ is a path of $\mathcal{P}$, $\langle n_k, \omega_k\rangle \models^B_\lambda \psi_2$ and $\langle n_i, \omega_i\rangle \models^B_\lambda \psi_1$ for every $0 \leq i < k$ where $\langle n_0, \omega_0\rangle = \langle n, \omega\rangle$, we show that there exists $k' \geq 0$ such that $\langle n'_0, \omega'_0\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_1, \omega'_1\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_2, \omega'_2\rangle ...\langle n'_{k'-1}, \omega'_{k'-1}\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_{k'}, \omega'_{k'}\rangle$ is a path of $\mathcal{P}'$, $\langle n'_{k'}, \omega'_{k'}\rangle \vdash^B_{\lambda'} \psi_2$ and $\langle n'_i, \omega'_i\rangle \vdash^B_{\lambda'} \psi_1$ for every $0 \leq i < k'$ where $\langle n'_0, \omega'_0\rangle = \langle n, \omega\rangle$. W.l.o.g., we suppose k is the minimum one.

    Since the abstraction adds one transition $\langle p, \gamma\omega\rangle \rightsquigarrow_{\mathcal{P}'} \langle p', \gamma'\omega\rangle$ for a sequence of transitions $\langle p, \gamma\omega\rangle \rightsquigarrow_{\mathcal{P}} \langle p_1, \gamma\omega\rangle \rightsquigarrow_{\mathcal{P}} \langle p_2, \gamma\omega\rangle, ..., \rightsquigarrow_{\mathcal{P}} \langle p_k, \gamma\omega\rangle \rightsquigarrow_{\mathcal{P}} \langle p', \gamma'\omega\rangle$, where $\gamma = \gamma'$, thus, there exists a subsequence $n_{i_0}, ..., n_{i_{k'}}$ of $n_0, ..., n_k$ such that $i_{k'} = k$, $i_0 = 0$ and

    $$\forall j : 0 \leq j \leq k',\ \langle n_{i_j}, \omega_{i_j}\rangle = \langle n'_j, \omega'_j\rangle.$$

    By applying the induction hypothesis, we obtain that $\langle n'_{k'}, \omega'_{k'}\rangle \vdash^B_{\lambda'} \psi_2$ and

    $$\forall j : 0 \leq j < k', \langle n'_i, \omega'_i\rangle \vdash^B_{\lambda'} \psi_1.$$

    Thus, $\langle n, \omega\rangle \vdash^B_{\lambda'} \psi$.

- Suppose there exists $k' \geq 0$ such that $\langle n'_0, \omega'_0\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_1, \omega'_1\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_2, \omega'_2\rangle ...\langle n'_{k'-1}, \omega'_{k'-1}\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_{k'}, \omega'_{k'}\rangle$ is a path of $\mathcal{P}'$, $\langle n'_{k'}, \omega'_{k'}\rangle \vdash^B_{\lambda'} \psi_2$ and $\langle n'_i, \omega'_i\rangle \vdash^B_{\lambda'} \psi_1$ for every $0 \leq i < k'$ where $\langle n'_0, \omega'_0\rangle = \langle n, \omega\rangle$, we show that there exists $k \geq 0$ such that $\langle n_0, \omega_0\rangle \rightsquigarrow_{\mathcal{P}} \langle n_1, \omega_1\rangle \rightsquigarrow_{\mathcal{P}} \langle n_2, \omega_2\rangle ...\langle n_{k-1}, \omega_{k-1}\rangle \rightsquigarrow_{\mathcal{P}} \langle n_k, \omega_k\rangle$ is a path of $\mathcal{P}$, $\langle n_k, \omega_k\rangle \models^B_{\lambda} \psi_2$ and $\langle n_i, \omega_i\rangle \models^B_{\lambda} \psi_1$ for every $0 \leq i < k$ where $\langle n_0, \omega_0\rangle = \langle n, \omega\rangle$.

  For every $0 \leq i \leq k'$: if $\langle n'_j, \omega'_j\rangle \rightsquigarrow_{\mathcal{P}'} \langle n'_{j+1}, \omega'_{j+1}\rangle$ is inspired by a transition rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta'$ which is added due to $\langle p, \gamma\rangle \rightsquigarrow_{\mathcal{P}} \langle p_1, \gamma\rangle \rightsquigarrow_{\mathcal{P}} \langle p_2, \gamma\rangle, ..., \rightsquigarrow_{\mathcal{P}} \langle p_k, \gamma\rangle \rightsquigarrow_{\mathcal{P}} \langle p', \gamma'\rangle$, according to the definition of the abstraction, we obtain that there exist $n^j_0, n^j_1, n^j_2, ..., n^j_m$ such that $n^j_0 = n'_j$, $n^j_m = n'_{j+1}$, $\omega'_{j+1} = \omega'_j$ and $\langle n^j_0, \omega'_j\rangle \rightsquigarrow_{\mathcal{P}} \langle n^j_1, \omega'_j\rangle, ..., \rightsquigarrow_{\mathcal{P}} \langle n^j_m, \omega'_j\rangle$.

  Otherwise we have $\langle n'_j, \omega'_j\rangle \rightsquigarrow_{\mathcal{P}} \langle n'_{j+1}, \omega'_{j+1}\rangle$.

  Since $n^j_0 \in \lambda'(b(a_1, ..., a_n))$ iff $n^j_i \in \lambda'(b(a_1, ..., a_n))$ for every $1 \leq i \leq m$ according to the definition of the abstraction and $\langle n'_j, \omega'_j\rangle \vdash^B_{\lambda'} \psi_1$, we obtain that $\langle n^j_i, \omega'_j\rangle \vdash^B_{\lambda'} \psi_1$ for every $0 \leq i \leq m$.

  By applying the induction hypothesis: we obtain that $\langle n^j_i, \omega'_j\rangle \models^B_{\lambda} \psi_1$ for every $0 \leq i \leq m$, and $\langle n'_{k'}, \omega'_{k'}\rangle \models^B_{\lambda} \psi_2$ where $\langle n'_{k'}, \omega'_{k'}\rangle = \langle n_k, \omega_k\rangle$.

  Thus, there exists $k \geq 0$ such that $\langle n_0, \omega_0\rangle \rightsquigarrow_{\mathcal{P}} \langle n_1, \omega_1\rangle \rightsquigarrow_{\mathcal{P}} \langle n_2, \omega_2\rangle ...\langle n_{k-1}, \omega_{k-1}\rangle \rightsquigarrow_{\mathcal{P}} \langle n_k, \omega_k\rangle$ is a path of $\mathcal{P}$, $\langle n_k, \omega_k\rangle \models^B_{\lambda} \psi_2$ and $\langle n_i, \omega_i\rangle \models^B_{\lambda} \psi_1$ for every $0 \leq i < k$ where $\langle n_0, \omega_0\rangle = \langle n, \omega\rangle$.

□

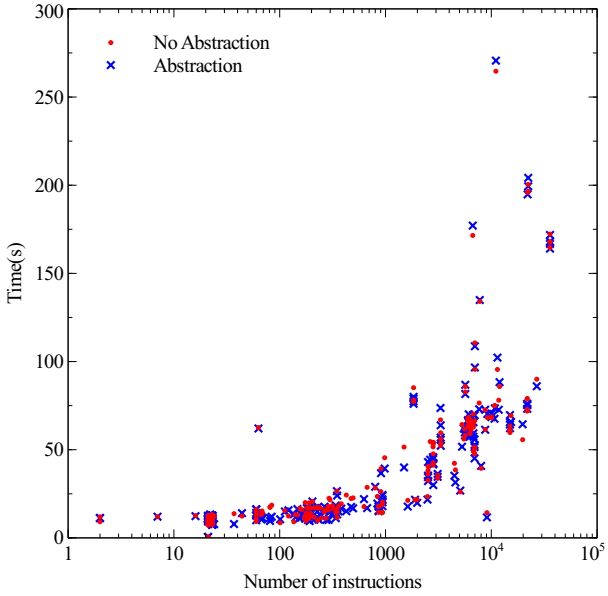

Fig. 8. Time Comparison

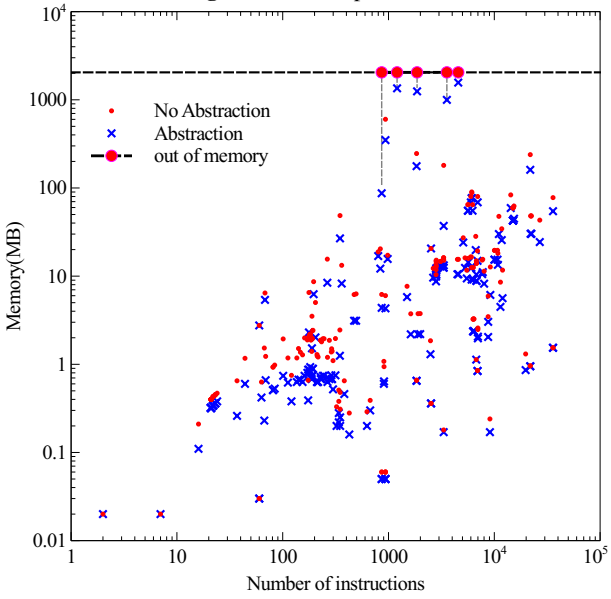

Fig. 9. Memory Comparison