Pushdown Model Generation of Malware

Nguyen Minh Hai[1], Mizuhito Ogawa[2] and Quan Thanh Tho[1]

[1]HoChiMinh City University of Technology, Vietnam
[2]Japan Advanced Institute of Science and Technology, Japan

Abstract. Model checking software consists of two steps: model generation and model checking. A model is often generated statically by abstraction, and sometimes refined iteratively. However, model generation is not easy for malware, since malware is often distributed without source codes, but as binary executables. Worse, sophisticated malware tries to obfuscate its behavior, like self-modification, which dynamically modifies itself and destination of indirect jumps.
This paper proposes a pushdown model generation of x86 binaries in an on-the-fly manner with concolic testing to decide the precise destinations of indirect jumps. A tool BE-PUM (Binary Emulation for PUshdown Model generation) is built on JakStab, and currently it covers 52 popular x86 instructions. Experiments are performed on 1700 malwares taken from malware database. Compared to JakStab and IDA Pro, two state-of-the-art tools in this field, BE-PUM shows better tracing ability, which sometimes shows significant differences.

Keywords: concolic testing, pushdown system, malware detection, binary code analysis, self-modifying code

1 Introduction

Malware Analysis *Malwares*, or *malicious softwares*, are computer programs which are intended to damage or disrupt a system. Popular kinds of malwares are classified as follows [1].

– *Virus*: It replicates possibly evolved copies when executed and then inserts those copies into other computer programs. *Worm* is a special kind of virus working on the network environment.
– *Trojan horse*: It comes as an infected program with attractive features. Once executed, a Trojan secretly sends private information to the hacker. Popular kinds include *Backdoor*, which allows remote connection without proper permission, and *Password-stealing*, which captures system passwords.
– *Spammer*: It sends unsolicited messages to a large group of users.
– *Flooder*: It attacks a computer with a heavy load of traffic, like distributed denial-of-service (DDoS) attack.
– *Keylogger*: It captures all of keystrokes on the victim computer, based on which the attacker can reveal its sensitive information.

They are distributed as binary executables, without source codes. There are three major techniques to detect malwares.

- Signature recognition.
- Virtual emulation in a sandbox.
- Program analysis.

Signature is a typical bit pattern, which characterizes malwares. Most of industrial malware detection methods depend on regular expression based signature recognition [1,2]. However, recent advanced obfuscation techniques and *polymorphic virus* show that they can evade signature recognition. For instance, a polymorphic virus can form a complex formal language [3], which are beyond regular expressions.

Advanced *obfuscation techniques* not only change the contents of the signatures but also the control flow. Common obfuscation techniques include:

- *Dead code insertion*: When replicated, it inserts a random block of codes that does not change the real behavior.
- *Code reordering*: It changes the location of procedures or changes the order of independent instructions within a procedure.
- *Register reassignment*: It changes registers used by live variables.
- *Instruction replacement*: It replaces instructions in the code by others with the same functions.

Typical techniques of *polymorphic virus* include mutation, e.g.,

- *Self-encryption*: The decryption module (often at the beginning) decrypts the rest of the code, by from simple XOR-ing to sophisticated ones.
- *Self-modification*: Typically, the destinations of indirect jumps are modified, including overwriting the return address in the stack.

For these advanced techniques, current approaches are either *virtual emulation* or *model checking*. Virtual emulation prepares a sandbox to explore behavior of malwares, which requires a deep encoding of system environments to emulate windows APIs [4]. This is not only heavy, but also not easy to find a suitable abstraction level. Furthermore, emulation may fail when malware changes its actions by probing the environment whether it is an emulator.

As an alternative, recent research attempts to infer an *abstract model* from binary executables. An abstract model commonly adopts the *control flow graphs* (CFG). Once a CFG (abstract model) is obtained, popular analysis techniques like model checking can be adopted [5,6,7,8,9,10].

However, such a model generation is not easy, since it requires disassembly, and obfuscation and mutation techniques confuse a lot. For instance, indirect jumps requires precise arithmetic analysis on 32-bit addresses and interpretation of x86 instructions to detect precise destinations. Such an analysis mutually depends on a model generation, and an on-the-fly model generation [11] is a typical technique. That is, starting from the entry, when an indirect jump is found, its destination is analyzed, and a partial model is enlarged. This continues until no more new destinations are found.

There are lots of binary analysis tools, e.g., CodeSurfer/x86 [12,13], McVeto [14], JakStab [15,16], BIRD [17], Renovo [18], Syman [19], and BINCOA/OSMOSE [20], among which CodeSufer/x86, McVeto, and JakStab apply static analysis, and BIRD, Renove, Syman, and BINCOA/OSMOSE apply concolic testing. Except for McVeto, they take a context-cloning (or context-insensitive) approach, and except for Syman, they do not support system calls. Especially, CodeSurfer is extended from a commercial product, known as IDA Pro[1], which is claimed to be one of the most popular and powerful tools for binary code analysis. However, it is also quite limited when dealing with indirect jumps.

Model-checking-based approaches for malware detection Among model generation approaches, model checking has been increasingly attracting much attention. Back to 2001, the idea of presenting binary code as a model and malicious behaviors as properties to be verified was proposed [21]. Recently, *pushdown model checking* [6] starts to be applied. In order to describe properties, LTL is firstly suggested [22]. Later on, variations of CTLs for describing malicious behavior are suggested, ranging from CTL [23], CTPL [9], SCTPL [6] and SCTPL/X [23]. Recent results in this approach have pointed out that a pushdown model is suitable for analysis of malware behaviors [5,6]. It is because viruses typically need to call system API to perform intended malicious actions.

Contribution This paper proposes a pushdown model generation (i.e., context-stacking approach) based on concolic testing, which is implemented as BE-PUM (Binary Emulation for PUshdown Model generation) as an extension of JakStab [15,16]. As our limited knowledge, BE-PUM solely generates pushdown models of binary codes including indirect jumps. Currently BE-PUM supports 52 popular x86 instructions (but does not support system calls), which covers more than 1700 malwares from *VX Heavens*[2] (consisting of 4123 malwares classified below). Experiments on 1700 malwares shows that BE-PUM outperforms JakStab and IDA Pro, sometimes significantly.

Kind	Virus	Backdoor	Email	P2P	Constr.	Exploit	IRC	VirTool	Net	Worm	IM
Number	2079	1079	359	105	86	85	73	68	66	64	59

2 Illustrating Example and Related Work

Illustrating Example In this section, we illustrate how BE-PUM works. For simplicity, we show Control Flow Graph (CFG) generation without procedure calls, which illustrates how to solve the destination of indirect jumps. The integration of CFG into a pushdown model will be formally discussed in the following

[1] `http://www.datarescue.com/idabase/`
[2] http://vx.netlux.org

```
start: ;Entry point
  0: cmp eax, 0
  1: jge l1
  2: jmp offset start
l1:
  3: cmp ebx, eax
  4: jge l2
  5: jmp offset start
l2:
  6: mov ecx, 4
  7: mul ecx
  8: mov eax, ebx
  9: cmp eax, 3
  10: je l3
  11: jmp offset start
l3:
  12: mov ebx, offset l4
  13: sub ebx, 3
  14: jmp ebx
l4:
  15: ret
```

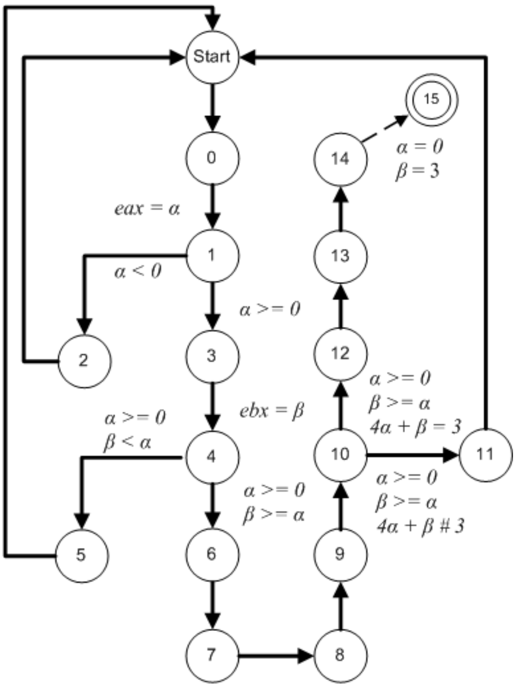

Fig. 1: CFG of an example code

sections. A target example is given in Fig. 1. A binary program starts at *start* and introduces an indirect jump at $L14$. The execution path leading to this dynamic jump is easily determined, i.e., $P = (star \to 0 \to 1 \to 3 \to 4 \to 6 \to 7 \to 8 \to 9 \to 10 \to 12 \to 13 \to 14)$. For an initial value α of the register *eax* and β of the register *ebx*, symbolic execution evaluates the path condition of P as $(\alpha >= 0) \wedge (\beta >= \alpha) \wedge (4 * \alpha + \beta = 3)$.

Then, these conditions are solved by an SMT solver to generate a test-case, say, $\alpha = 0$ and $\beta = 3$. By emulating the program with them, finally $L15$ is discovered as a new target of the indirect jump at $L14$. The resulting CFG is shown in Fig. 1, where the dotted arrow indicates a newly generated edge. Fig. 2 shows the analysis results of JakStab, IDA Pro, and BE-PUM, from left to right. JakStab and IDA Pro fail to detect the destination of *jmp* at $L14$, while BE-PUM successfully generates an edge.

Related Work There are various model generation tools from binary executables, e.g., BINCOA/OSMOSE [20,24], CodeSurfer/x86 [12,13], McVeto [14], JakStab [15,16], BIRD [17], Syman [19], and Renovo [18].

Among these tools, BIRD has focus more on disassembly. Although all of them applies disassembly (mostly IDA Pro is used as a preprocessor) and in-

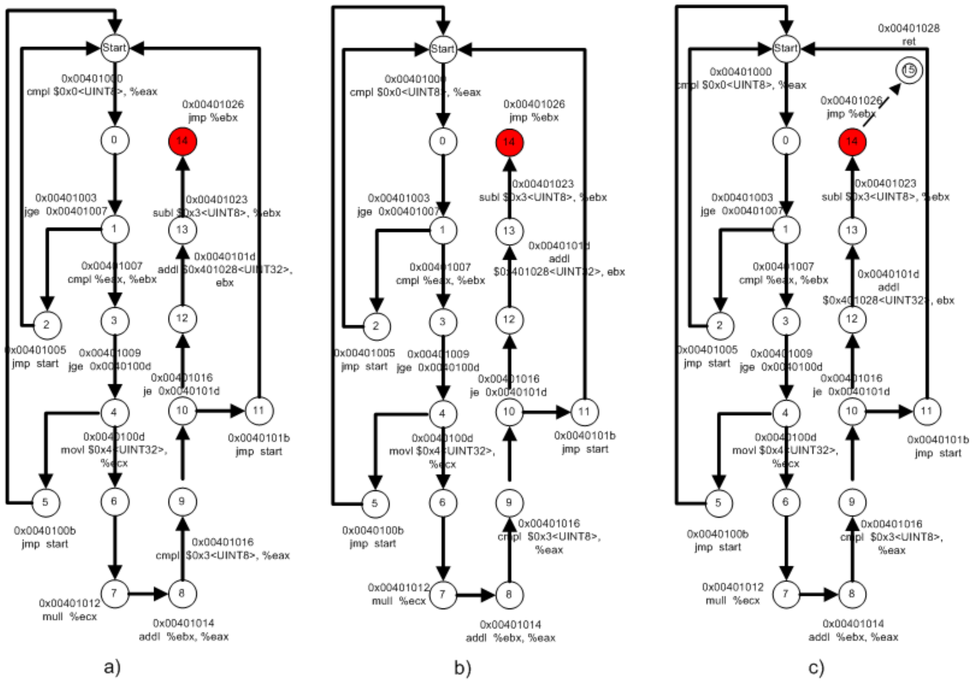

Fig. 2: Generated results by (a) JakStab, (b) IDA Pro and (c) BE-PUM

terpretation is given at assembly level, OSMOSE and CodeSurfer/x86 support 32-bit vector models, which directly describe memory as a state. For instance, OSMOSE is based on a DBA (Dynamic Bit-vector Automaton) [25].

Among above mentioned difficulties, self-decryption and system calls have extra hardness, and few tools can handle them. For self-decryption, Polyunpack [26] and Renovo [18] are such examples, in which differences between static codes and dynamic codes detect malicious codes. For system calls, only Syman supports with Windows API emulator Aligator [4].

For handling indirect jumps, detection of their destinations requires precise arithmetic analysis on 32-bit addresses and interpretation [27] of x86 instructions. We have three axes to classify tools.

- *Whether static or dynamic analyses*: CodeSurfer/x86, McVeto, and JakStab apply static analyses, whereas BIRD, Renovo, Syman, and BINCOA/OSMOSE apply dynamic emulation (except for Syman which also apply concolic testing). BE-PUM belongs to the latter.
- *Whether an on-the-fly model generation [11]*: JakStab, McVeto, Syman, and BINCOA/OSMOSE apply an on-the-fly modeling. BE-PUM uses the same method. CodeSurfer/x86 applies a static analysis (value-set analysis) first and then generates a CFG.

– *Context-stacking vs context-cloning*: Except for McVeto, they adopt context-cloning (or context-insensitive) approaches. McVeto also applies CEGAR-like abstraction refinement.

3 Preliminaries

3.1 Pushdown Systems

For a context-sensitive model, there are two approaches: context-cloning and context-stacking. We focus on malware that does not modify intermediate stack frames, but may modify the top stack frame (i.e., return address / value), and apply a pushdown system.

Definition 1. *A* pushdown system (PDS) *is a triplet* $\langle P, \Gamma, \Delta \rangle$ *where*

– P *is a finite set of states,*
– Γ *is finite stack alphabet, and*
– $\Delta \subseteq P \times \Gamma^{\leq 2} \times P \times \Gamma^{\leq 2}$ *is a finite set of transitions, where* $(p, v, q, w) \in \Delta$ *is denoted by* $(p, v \rightarrow q, w)$*.*

We use $\alpha, \beta, \gamma, \cdots$ to range over Γ, and $w, v, \cdots$ over words in Γ^*. A *configuration* $\langle p, w \rangle$ is a pair of a state p and a stack content (word) w. As convention, we denote configurations by $c_1, c_2, \cdots$. One step transition $\hookrightarrow$ between configurations is defined as follows. $\hookrightarrow^*$ is the reflexive transitive closure of $\hookrightarrow$.

$$\frac{\langle p, \gamma w \rangle \hookrightarrow \langle p', \gamma' w \rangle}{(p, \gamma \rightarrow p', \gamma') \in \Delta}\ \textit{inter} \quad \frac{\langle p, \gamma w \rangle \hookrightarrow \langle p', \alpha\beta w \rangle}{(p, \gamma \rightarrow p', \alpha\beta) \in \Delta}\ \textit{push} \quad \frac{\langle p, \gamma w \rangle \hookrightarrow \langle p', w \rangle}{(p, \gamma \rightarrow p', \epsilon) \in \Delta}\ \textit{pop}$$

A PDS enjoys decidable *configuration reachability*, i.e., given configurations $\langle p, w \rangle$, $\langle q, v \rangle$ with $p, q \in P$ and $w, v \in \Gamma^*$, decide whether $\langle p, w \rangle \hookrightarrow^* \langle q, v \rangle$.

3.2 Concolic Testing

Concolic testing is a hybrid software verification technique that combines concrete execution with *symbolic execution* [28], which is available in testing tools like PathCrawler [29], jCUTE [30], and SAGE [31]. As compared to traditional white-box testing, concolic testing can reduce test data generation by restricting attention to feasible execution paths.

Let us consider the example in Fig. 1. For traditional white-box testing, there would be 4 path conditions $eax < 0$, $(eax >= 0) \wedge (ebx < eax)$, $(eax >= 0) \wedge (ebx >= eax) \wedge (4 * eax + ebx \neq 3)$, and $(eax >= 0) \wedge (ebx >= eax) \wedge (4 * eax + ebx = 3)$ needed to be considered. When concolic testing is applied, it first randomly generates values for eax and ebx, e.g. $eax = 1$ and $ebx = 2$. In the concrete execution, Line 1 is reached since the condition of $eax >= 0$ is true. Line 4 also holds the condition $ebx >= eax$, but Line 10 fails to hold $4 * eax + ebx = 3$. Concurrently, the symbolic execution follows the same path, but treating eax and ebx as symbolic variables. The condition $(eax >= 0) \wedge (ebx >= eax) \wedge (4 * eax + ebx \neq 3)$ is a *path condition*. To follow a different execution path on the next run, the reason $(4 * eax + ebx \neq 3)$ of failure (at Line 10) is negated as $(4 * eax + ebx = 3)$. An SMT solver is then invoked to find values satisfying $(eax >= 0) \wedge (ebx >= eax) \wedge (4 * eax + ebx = 3)$, e.g., $eax = 0$, $ebx = 3$. Its execution reaches Line 12.

In our context of malware model generation, concolic testing is applied to decide the destination address when indirect jumps are encountered. Note that this stepwise execution requires virtual emulation.

4 X86 Binary Execution Models

4.1 Memory Models and x86 operational semantics

In this section, we present an abstract memory model, on which operational semantics of x86 binary is given. Our semantics is inspired by [27], but for a direct connection with our binary emulation, everything except for 9 system flags (i.e., R, S, M in Definition 2) is represented by 32-bit vectors, and arithmetic operations are bit-encoded on these vectors.

We assume that a target X86 binary program $Prog_{x86}$ is loaded and consumes in a bounded area of memory, referred as M. The instruction pointer eip is a special register that points to the current address of instructions, and it is initially set to the entry address of $Prog_{x86}$.

Definition 2. *A* memory model *is a tuple (F, R, S, M), where F is the set of 9 system flags (AF, CF, DF, IF, OF, PF, SF, TF, and ZF), R is the set of 16 registers (eax, ebx, ecx, edx, esi, edi, esp, edp, cs, ds, es, fs, gs, ss, eip, and $eflags$), M is the set of memory locations to store, and $S(\subseteq M)$ is the set of contiguous memory locations for a stack (associated standard push/pop operations).*

Let $k = Env_R(eip) \in M$ be a mapping $instr(Env_M, k)$ that disassembles a binary code at the memory location k and return an instruction (with its arguments). An operational semantics *of a binary code $Prog_{x86}$ is described as transitions (in Fig. 3) among environments Env, which consists of a flag valuation Env_F, a register valuation Env_R, a stack valuation Env_S, and a memory valuation Env_M (on $M \setminus S$).*

Remark 1. The reason to define operational semantics directly on a binary executable is that self-modifying codes may not have statically corresponding assemblers. For instance, execution of the head of a self-decryption virus decrypts the rest, say by XOR-ing. Thus, the latter part does not have corresponding assembly code before decryption.

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) ='' add\ r_1\ r_2'', \\ w = Env_R(r_1) + Env_R(r_2), m = k + |add\ r_1\ r_2| \end{array}}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m, r_1 \leftarrow w], Env_S, Env_M)}\ [Addition]$$

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) = \text{``}sub\ r_1\ r_2'', \\ w = Env_R(r_1) - Env_R(r_2), m = k + |sub\ r_1\ r_2| \end{array}}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m, r_1 \leftarrow w], Env_S, Env_M)}\ [Subtraction]$$

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) ='' and\ r_1\ r_2'', \\ w = Env_R(r_1) \wedge Env_R(r_2), m = k + |and\ r_1\ r_2| \end{array}}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m, r_1 \leftarrow w], Env_S, Env_M)}\ [And]$$

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) = \text{``}or\ r_1\ r_2'', \\ w = Env_R(r_1)|Env_R(r_2), m = k + |or\ r_1\ r_2| \end{array}}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m, r_1 \leftarrow w], Env_S, Env_M)}\ [Or]$$

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) ='' mov\ t\ r'', \\ r \in R, w = Env_R(r), m = k + |mov\ t\ r| \end{array}}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m], Env_S, Env_M[t \leftarrow w])}\ [Move]$$

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) ='' xchg\ r_1\ r_2'', \\ w_1 = Env_R(r_1), w_2 = Env_R(r_2), m = k + |xchg\ r_1\ r_2| \end{array}}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m, r_1 \leftarrow w_2, r_2 \leftarrow w_1], Env_S, Env_M)}\ [Exchange]$$

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) ='' call\ r'', \\ m' = k + |call\ r|, m = Env_R(r), push(S, m') = S' \end{array}}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m], Env_{S'}, Env_M)}\ [Call]$$

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) ='' cmp\ r_1\ r_2'', m = k + |cmp\ r_1\ r_2|, \\ c = Env_R(r_1) - Env_R(r_2), sf = (c < 0), zf = (c = 0), \\ cf = ((Env_R(r_1) >= 0) \wedge (Env_R(r_2) < 0)) \vee ((c < 0) \wedge ((Env_R(r_1) >= 0) \vee (Env_R(r_2) < 0)), \\ of = ((Env_R(r_1) < 0) \wedge (Env_R(r_2) >= 0) \wedge (c > 0)) \vee ((Env_R(r_1) >= 0) \wedge (Env_R(r_2) < 0) \wedge (c < 0)) \end{array}}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F[CF \leftarrow cf, OF \leftarrow of, SF \leftarrow sf, ZF \leftarrow zf], Env_R[eip \leftarrow m], Env_S, Env_M)}\ [Cmp]$$

$$\frac{Env_R(eip) = k, instr(Env_M, k) ='' ret'', empty(S)}{(Env_F, Env_R, Env_S, Env_M) \to \perp}\ [Return]$$

$$\frac{Env_R(eip) = k, instr(Env_M, k) ='' ret'', \neg empty(S), pop(S) = (S', m)}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m], Env_{S'}, Env_M)}\ [Return]$$

$$\frac{Env_R(eip) = k, instr(Env_M, k) ='' jmp\ r'', Env_R(r) = m}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m], Env_S, Env_M)}\ [(Indirect)Jump]$$

$$\frac{R(eip) = k, instr(Env_M, k) ='' jmp\ m'', M(m) = m'}{(Env_F, Env_R, Env_S, Env_M) \to (Env_F, Env_R[eip \leftarrow m'], Env_S, Env_M)}\ [Jump]$$

Fig. 3: The rules of operational semantics

4.2 Pushdown model

A control flow graph (CFG) is often intra-procedural. We can consider a call graph and/or an inter-procedural CFG. Our choice is a pushdown model as a unified representation of a call graph and an intra-procedural CFG.

We assume that self-modification on a stack occurs only for the return address, i.e., the top stack frame only (not at an intermediate stack frame), and targets only on sequential binaries. These assumptions validate a pushdown model.

A pushdown model of $Prog_{x86}$ is given as transitions among pairs (k, asm) of memory locations $k (\in M)$ and corresponding assembly instructions asm. Such an assembly instruction asm is obtained by disassembly of a binary sequence starting from k, and we refer by $asm = instr(Env_M, k)$.

Definition 3. *Let $P = \{(k, asm) \mid k \in M,\ asm \text{ is an x86 assembly instruction}\}$. A pushdown model $\mathcal{P}$ of an x86 binary program $Prog_{x86}$ is a tuple $\langle P, P, \Delta\rangle$, where $\Delta \subseteq (P \times P) \times (P \times P^{\leq 2})$. For a transition of x86 operational semantics $(Env_F, Env_R, Env_S, Env_M) \to (Env'_F, Env'_R, Env'_S, Env'_M)$ with $k = Env_R(eip)$ and $k' = Env'_R(eip)$, we have*

- **Push rules** $\langle instr(Env_M, k), \epsilon\rangle \hookrightarrow \langle instr(Env'_M, m), instr(Env'_M, w)\rangle$, *corresponding to Call rule in Fig. 3.*
- **Pop rules** $\langle instr(Env_M, k), instr(Env_M, m)\rangle \hookrightarrow \langle instr(Env'_M, m), \epsilon\rangle$, *corresponding to Return rules in Fig. 3.*
- **Internal rules** $\langle instr(Env_M, k), \epsilon\rangle \hookrightarrow \langle instr(Env'_M, m), \epsilon\rangle$, *corresponding to other rules in Fig. 3.*

Note that $asm = instr(Env_M, k)$ can be different even for the same k, since Env_M can be modified. When such self-modification occurs, we distinguish (k, asm) and (k, asm') as different states. If there are no self-modification, we often identify P with the set of program locations of a corresponding x86 assembly program. A pushdown model extracts only control structures by omitting the environment Env from the operational semantics. Thus, a pushdown model will have nondeterministic transitions, e.g., at conditional branches and depending on system flag status.

Example 1. A pushdown model of the program in Fig. 1 is a tuple $\langle P, P, \Delta\rangle$ with $P = \{(L1, \text{"}\texttt{jge l1}\text{"}), \ldots, (L15, \texttt{ret})\}$. Δ is the set of pushdown transition rules corresponding to instructions in P. In this example, there are no calls and the stack does not change. Except for $(L14,\ \texttt{jmp ebx})$, the pushdown transition rules follow in a straightforward way, since their next instructions are statically decided. For instance, at $L6$,

$$\langle (L6,\ \text{"}\texttt{mov ecx, 4}\text{"}), \epsilon\rangle \hookrightarrow \langle (L7,\ \text{"}\texttt{mul ecx}\text{"}), \epsilon\rangle$$

where in the execution model, ecx is updated to 4.

At the jump instruction at $L1$ has nondeterministic transition rule

$$\langle (L1,\ \text{"}\texttt{jge l1}\text{"}), \epsilon\rangle \hookrightarrow \langle (L2,\ \text{"}\texttt{jmp offset start}\text{"}), \epsilon\rangle$$
$$\langle (L1,\ \text{"}\texttt{jge l1}\text{"}), \epsilon\rangle \hookrightarrow \langle (L3,\ \text{"}\texttt{cmp ebx, eax}\text{"}), \epsilon\rangle$$

corresponding to two possible signs of eax. Transition rules at $L14$ are quite complicated, and will be explained in Example 2.

5 Pushdown Model Generation by Concolic Testing

5.1 On-the-fly model generation by concolic testing

The aim of our tool BE-PUM (Binary Emulation for PUshdown Model generation) is to precisely handle indirect jumps, which requires correct arithmetic analysis on 32-bit addresses and interpretation of x86 instructions (Section 4). Such an analysis mutually depends on a model generation, which is sometimes said as a "*chicken and egg*" problem. The situation is similar to context-sensitive points-to analysis of Java.

Destination of a method invocation in Java is decided by dynamic types. On-the-fly point-to analysis mutually generates and checks a partial model [32]. That is, starting from the program entry, when method invocation is found, dynamic types are statically analyzed. We apply a similar on-the-fly model generation, replacing a method invocation with an indirect jump.

Malware is usually much smaller than web applications in Java, and its complex behavior requires more precise arithmetic analysis on addresses. For such an analysis on a partial model, we have choices, static analysis and dynamic analysis. Our choice is a dynamic method, concolic testing. Currently, we apply concolic testing for all instructions to decide next locations. Although we admit a room for optimization by avoiding concolic testing for immediate instructions, this is also effective for obfuscation by *opaque* predicates, e.g., conditions like $x^2 \geq 0$.

Definition 4 presents the rules for on-the-fly model generation, which is a saturation procedure on configurations of the form $\langle P, \Delta, \psi \rangle$. $\psi(k, asm)$ is the path precondition at (k, asm), which initially set **true** at the entry (k_0, asm_0).

Definition 4. *The initial configuration is $\langle \{(k_0, asm_0)\}, \phi, \psi_0 \rangle$, where (k_0, asm_0) is the pair of the entry address $k_0(\in M)$ and the initial instruction asm_0, and*

$$\psi_0(k, asm) = \begin{cases} \textbf{true} \ \textit{if} \ (k, asm) = (k_0, asm_0). \\ \textbf{false} \ \textit{otherwise.} \end{cases}$$

For $(k, asm) \in P$, let $(m, asm') = next(k, asm)$, which is obtained by the transition $Env \to Env'$ (described in Fig. 3) such that Env satisfies the constraint $\psi(k, asm)$, $k = Env_R(eip)$, and $asm = instr(Env_M, k)$.[3]

When (m, asm') is not a system call, the on-the-fly model generation continues with rules $\langle P, \Delta, \psi \rangle \vdash \langle P', \Delta', \psi' \rangle$ such that

- $P' := P \cup \{(m, asm')\}$,
- $\Delta' := \Delta \cup \{rule\}$ *for rule described in Definition 3, and*

[3] $Env \to Env'$ is executed by concolic testing on a binary emulator.

– $\psi'(m, asm') := \psi(m, asm') \vee (SideCond \wedge post(\psi(k, asm)))$ *for the side conditions* $SideCond$ *appearing in* $Env \rightarrow Env'$ *and the strongest post condition* $post(\psi(k, asm))$ *of* $\psi(k, asm)$.

Note that valuation of system flags Env_F can be any Boolean combination, since windows OS can generate all of them. This leads to non-deterministic transitions at x86 instructions, like `cmp` and `cjump`[4].

The on-the-fly pushdown model generation continues until P and Δ converge (regardless of convergence on ψ[5]).

Since we apply concolic testing (implemented with an SMT solver on linear integer arithmetic) to decide the next instruction at each configuration, the generated pushdown model is an under approximation of concrete execution.

Theorem 1. *For an X86 binary program* $Prog_{x86}$ *with the entry* (k_0, asm), *if* $((k_0, asm), \epsilon) \hookrightarrow^* ((m, asm'), stack)$ *in the model* $\mathcal{P}$, *there exist* $Env = (Env_F, Env_R, Env_S, Env_m)$ *and* $Env' = (Env'_F, Env'_R, Env'_S, Env'_m)$ *such that*

$Env \rightarrow^* Env'$, $Env_R(eip) = k_0$, $Env'_R(eip) = m$, $instr(Env_M, k_0) = asm$, $instr(Env'_M, m) = asm'$, *and* Env'_S *describes stack.*

Example 2. We will follow Example 1. A transition rule at $L14$ is an indirect jump. The path precondition $\psi(L14,\ ''\texttt{jmp ebx}'')$ is

$$(\alpha >= 0) \wedge (\beta >= \alpha) \wedge (4 * \alpha + \beta = 3)$$

for the initial values α and β of the registers *eax* and *ebx*, respectively. A satisfiable instance is $\alpha = 0$ and $\beta = 3$. Testing (on a binary emulator) with them leads to $next(L14,\ ''\texttt{jmp ebx}'') = (L15,\ ''\texttt{ret}'')$, and we obtain $(L15,\ ''\texttt{ret}'')$ and $\langle (L14,\ ''\texttt{jmp ebx}''), \epsilon \rangle \hookrightarrow \langle (L15,\ ''\texttt{ret}''), \epsilon \rangle$ for updates of P and Δ, respectively.

There are no more transitions to add, and the model generation converges.

5.2 BE-PUM implementation

BE-PUM is built on JakStab [15,16], which implements Definition 4. Compared with JakStab,

– BE-PUM applies concolic testing to decide the destination of indirect jumps, whereas JakStab applies a static analysis.
– BE-PUM takes a context-stacking approach, whereas JakStab applies a context-cloning approach.

Fig. 4 shows the architecture of BE-PUM, which consists of three components: *intra-procedural CFG extension*, *path condition solving*, and *binary emulation*.

Intra-procedural CFG extension is by (modified) JakStab. The modification is, when an indirect jump is reached, instead of a static analysis (default for JakStab), BE-PUM interrupts JakStab and passes to the path condition solving.

[4] Currently, BE-PUM implements only the former.
[5] The convergence of ψ requires the invariant generation, which is beyond the scope.

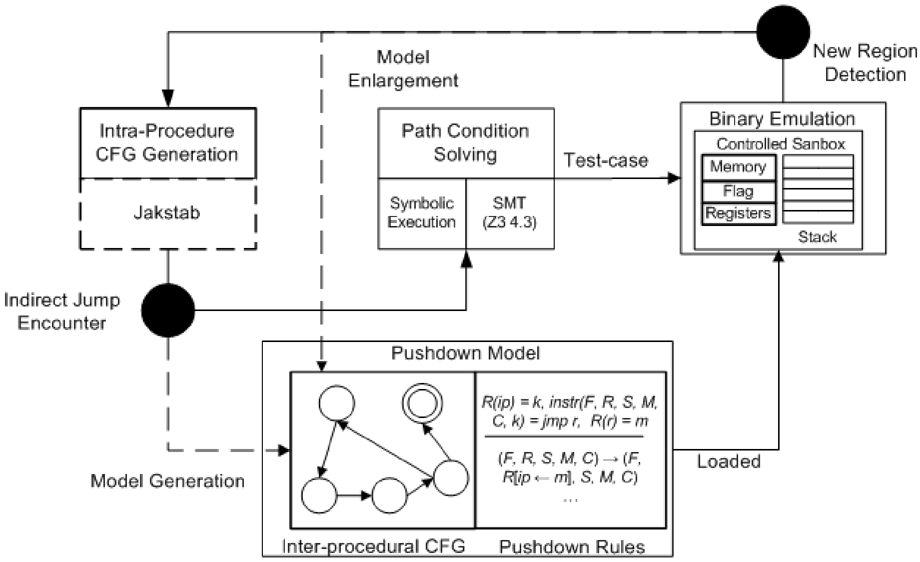

Fig. 4: The BE-PUM architecture

Note that a step-wise concolic testing requires both path condition solving and virtual emulation, If concolic testing detects an unexplored area of codes, we enlarge a pushdown model accordingly. Then, the control of BE-PUM returns to the intra-procedure CFG extension component. BE-PUM will terminate if either the exploration has converged, or reaching to system calls. The latter limitation comes from our current binary emulation (and the execution model) does not cover Windows APIs.

Path condition solving BE-PUM applies *symbolic execution* to evaluate the path conditions (in linear arithmetic) of a pushdown model, and adopts *Z3.4.3*[6] to solve the path conditions. Then, a satisfiable instance is an input of concolic testing on the binary emulator.

Binary Emulation BE-PUM prepares a controlled sandbox, which implements the x86 operational semantics (Definition 2). All elements R, S, M except for Boolean system flags F are implemented as 32-bit vectors.

Since the total number of x86 instructions is about one thousand, we first focus on frequently used instructions for implementation of BE-PUM. Table 1 shows the number of malwares that contains a specified instruction appears in the malware database *VX Heavens* (Section 2). BE-PUM covers 52 instructions (in Table 2), which are selected by frequency in the malware database *VX Heavens* and can cover more than 1700 malwares.

[6] `http://z3.codeplex.com`

Table 1: Popular x86 instructions in malware

Instruction	push	mov	jmp	dec	pop	call	add	inc	xor	sub	je	jne	cmp
Occurrences	2974	2756	2590	2547	2469	2282	2155	2089	2037	1771	1707	1618	1607
Instruction	or	jb	jae	lea	and	jbe	ja	ret	imul	shl	xchg	jo	ror
Occurrences	1460	1418	1313	1163	1151	1042	953	894	851	709	660	612	529

Table 2: List of supported x86 instructions

Arithmetic		**Call**	**Conditional Jump**				**Jump**	**Move**	**Return**	**Control**
add	sub	call	je	jnz	jc	jnc	jmp	mov	ret	cmp
and	or		jle	jnge	jge	jnle		int		push
xor	adc		js	jz	jb	jnb		lea		pop
imul	sal		jbe	jng	ja	jnl		xchg		nop
shl	shr		jo	jns	jne	jnae				test
inc	dec		jl	jnbe	jae	jna				
rol	ror		jg	loop						

Remark 2. Preliminary BE-PUM [33] supported only 18 instructions,

- arithmetic instructions (add, sub, shr, shl, dec, inc),
- logic instructions (and, or), jump instruction (jmp),
- conditional jump instructions (je, jle, ja, jne, jge, jng),
- move instruction (mov, lea), and compare instruction (cmp),

but not procedure calls (call and ret). Thus, it ran only for small toy examples.

6 Experiments

All experiments are performed Windows XP on AMD Athlon II X4 635 Processor with 2.9 GHz and 8 GB of memory. Though our ideas on pushdown model generation can be applied to self-modifying malwares, our experiments are on malwares with indirect jumps only.

6.1 Checking Accuracy on 4 Malwares with Source Code

For checking the accuracy of our method, Table 3 shows the experimental results on 4 viruses, *DeadKennedy*, *Pony*, *Triv_216*, and *Insert*, whose source codes are available. We manually inspected on their source codes to confirm the accuracy of generated CFGs.

In Table 3, the columns *Inst* are the number of instructions in the original code, which are detected by JakStab, IDA Pro, and BE-PUM, respectively. The Columns *Cvrg* show the coverage by JakStab, IDA Pro, and BE-PUM, respectively. All of them terminate when either they cannot explore further or they reach to system calls.

Table 3: Experimental results

Program		JakStab				IDA Pro				BE-PUM			
Name	Inst	Time(ms)	Inst	Edges	Cvrg	Time(ms)	Inst	Edges	Cvrg	Time(ms)	Inst	Edges	Cvrg
Dead Kennedy	200	800	97	101	48.5%	853	100	102	50%	18580	108	112	54%
Insert	173	200	44	46	25.4%	254	44	46	25.4%	2390	50	53	28.9%
Triv_216	102	160	19	19	18.6%	210	21	21	20.6%	1470	49	50	48%
Pony	758	180	35	35	4.61%	230	40	42	5.5%	24490	129	135	17%

Fig. 5 compares the generated models on *Pony*. BE-PUM outperforms JakStab and IDA Pro, at the cost of computational time.

The improvement comes from precision on indirect jumps. For example, when reaching to `jmp eax`, JakStab fails to evaluate the value of `eax`, and its disassembly fails. BE-PUM applies concolic testing, and successfully disassembles.

BE-PUM stops when reaching to system calls, e.g., *FindWindowA, PeekMessageA* in *user32.dll*, and *GetModuleFileNameA* in *kernel32.dll*. At the moment, our binary emulation does not cover them, which requires Windows API emulation as Syman [19] does.

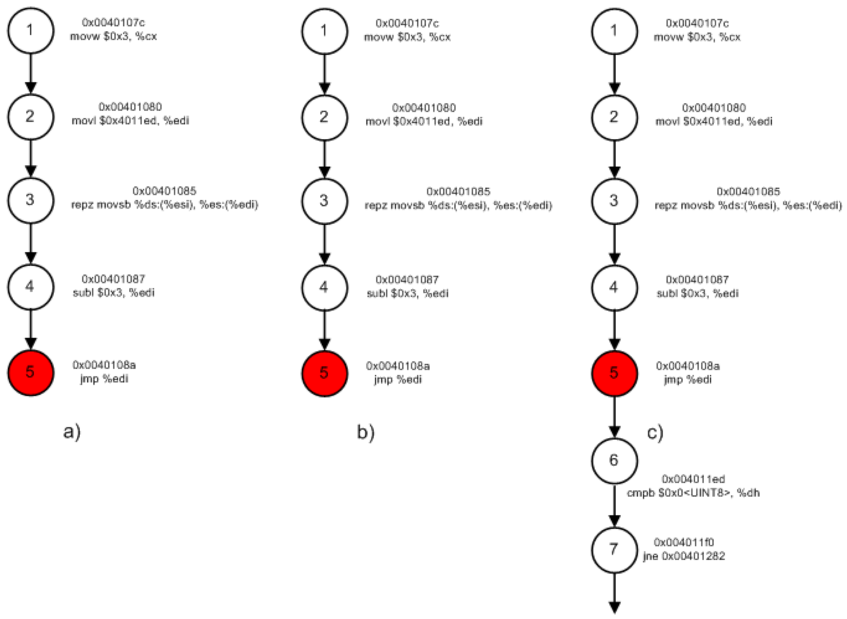

Fig. 5: CFG generated by (a) JakStab, (b) IDA Pro, and (c) BE-PUM

6.2 Experiments on 1700 Malwares without Source Code

For comparison with JakStab and IDA Pro, BE-PUM is tested on 1700 malwares, which are covered in our selected 52 most popular instructions. Comparison is

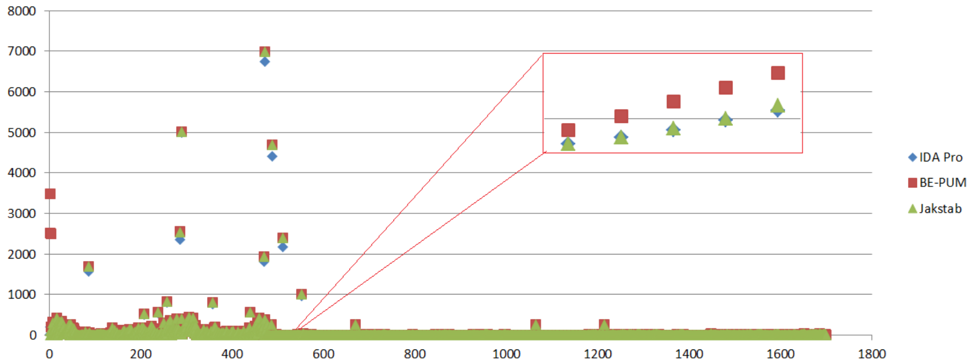

Fig. 6: Comparison of reachable nodes between JakStab, IDA Pro, and BE-PUM

summarized as a graph in Fig. 6. In Fig. 6, the x-axis presents the virus number and the y-axis describes the number of reachable nodes.

Table 4: Some Results of CFG construction

Example	JakStab			IDA Pro			BE-PUM		
	Time(ms)	Nodes	Edges	Time(ms)	Nodes	Edges	Time(ms)	Nodes	Edges
Constructor.Win32.Agent.a	169	176	177	131	171	173	1313	183	184
Rootkit.Win32.Agent.bd	1511	311	319	1112	301	303	15266	328	336
Rootkit.Win32.Agent.h	552	327	349	755	326	347	3703	332	354
Virus.DOS.Abraxas.1881	90	64	70	123	63	68	3860	69	75
Virus.DOS.Fisher.2420	387	120	123	427	124	128	5953	128	131
Virus.DOS.HLLO.Harakiri.5488	527	255	259	485	235	243	5516	259	263
Virus.DOS.HLLO.Horney	193	184	188	252	164	172	7281	190	194
Virus.DOS.HLLP.Arjinf.7598	4	13	12	45	22	23	2359	146	166
Virus.DOS.HLLP.Colba.7981	80	106	106	103	105	106	3266	3477	3479
Virus.DOS.HLLP.DarkFox.4997	13	9	8	33	14	15	8484	20	20
Virus.DOS.HLLP.Rock.8875	25	13	12	46	21	22	48047	200	225
Virus.DOS.Shish.1142	35	8	8	17	2	2	25797	2507	2514
Virus.DOS.SillyRC.291.a	5	31	30	37	28	29	9328	38	38
Virus.DOS.Slowly.1249	12	8	8	39	18	18	28422	2529	2536
Virus.DOS.Small.118	8	29	29	43	25	28	12500	47	49
Virus.DOS.Tiny.146	594	423	423	485	410	415	13453	429	430
Virus.DOS.Tiny.154	17	27	28	39	27	28	2344	34	35
Virus.DOS.Tiny.156	19	27	28	45	27	28	2375	34	35
Virus.DOS.Tiny.158	19	27	28	36	27	28	2375	34	35
Virus.DOS.Tiny.320	46	54	54	20	2	2	1421	120	121
Virus.DOS.Trebujena.1094	3	11	10	15	2	2	1063	16	15

Table 4 extracts some examples from Fig. 6. In Table 4, the columns *Time* show the computational time in milliseconds of JakStab, IDA Pro, and BE-PUM, respectively. The columns *Nodes* and *Edges* are the numbers of instructions and edges, respectively.

BE-PUM detects more nodes and edges compared to JakStab and IDA Pro. Some (like HLLP.Colba.7981, Slowly.1249, Shish.1142, HLLP.Rock.8875, HLLP.Arjinf.7598, and Tiny.320 in Table 4) shows significant improvements.

The reason is that these viruses hide large branches after indirect jumps, which can only detected by BE-PUM.

7 Conclusion and Future Work

This paper proposed an on-the-fly pushdown model generation of x86 binaries with concolic testing to decide the precise destinations of indirect jumps. Experiments were performed on 1700 malwares taken from malware database. Compared to JakStab and IDA Pro, two emerging tools in academic and industry communities, BE-PUM shows better tracing ability, which sometimes shows significant differences.

Often among existing tools, too large models are generated, partially because they are applying either context-cloning or context-insensitive approach. That is, when procedure calls occurs, they simply extend a CFG and connect edges to locations in disassembled codes, whereas context-cloning copies a location and context-insensitive approach does not. Our modeling follows context-stacking and generates a pushdown model.

There are lots of future work.

- *Pushdown model checking*: SCTPL and SLTPL (which are variants of CTL and LTL, respectively) pushdown model checking are applied to find suspicious system calls [6,7]. They rely on IDA Pro for disassembly, and they cannot handle indirect jumps and self-modifying code. We are planning to apply weighted pushdown model checking [34] on the result of BE-PUM. Although model checking is often an over-approximation (and our model generation is an under-approximation as stated in Theorem 1), we hope that our method is precise in practice.
- *Self-decryption*: Few model generation tools supports self-decryption, such as Polyunpack [26] and Renovo [18]. Currently BE-PUM implementation does not cover self-modification and self-decryption, due to technical reasons on the use of JakStab.
- *Windows system calls*: Few model generation tools supports system calls, such as Syman [19], which applies light-weight Windows API emulator Alligator. We are considering an alternative stub-based approach.

Sometimes, BE-PUM terminates with unknown jump destinations. We expect that they are obtained by using Windows API `GetProcAddress` in advance or accessing the memory address of *kernel32.dll*, but still under investigation.

References

1. P. Szor, *The Art of Computer Virus Research and Defense.* Addison-Wesley Professional, 2005.
2. E. Filiol, “Malware pattern scanning schemes secure against black-box analysis,” *Journal in Computer Virology*, vol. 2, pp. 35–50, 2006.

3. E. Filiol, "Metamorphism, formal grammars and undecidable code mutation," *International Journal of Computer Science*, vol. 2, pp. 70–75, 2007.
4. A. Mori, T. Izumida, T. Sawada, and T. Inoue, "A tool for analyzing and detecting malicious mobile code," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 831–834, 2006. LNCS 3233.
5. G. Balakrishnan, T. W. Reps, 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 *17th Computer Aided Verification (CAV 2005)*, pp. 158–163, 2005. LNCS 3576.
6. F. Song and T. Touili, "Pushdown model checking for malware detection," in *18th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 110–125, 2012. LNCS 7214.
7. F. Song and T. Touili, "LTL model-checking for malware detection," in *19th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 416–431, 2013. LNCS 7795.
8. A. Holzer, J. Kinder, and H. Veith, "Using verification technology to specify and detect malware," in *11th Computer Aided Systems Theory (EUROCAST)*, pp. 497–504, 2007. LNCS 4739.
9. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *2nd Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2005)*, pp. 174–187, 2005. LNCS 3548.
10. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, pp. 424–438, 2010.
11. J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham, "Detection of injected, dynamically generated, and obfuscated malicious code," in *Proc. 2003 ACM Workshop on Rapid Malcode (WORM)*, pp. 76–82, 2003.
12. G. Balakrishnan and T. W. Reps, "Analyzing memory accesses in x86 executables," in *13th Compiler Construction (CC 2004)*, pp. 5–23, 2004. LNCS 2985.
13. G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum, "Codesurfer/x86-a platform for analyzing x86 executables," in *14th Compiler Construction (CC 2005)*, pp. 250–254, 2005. LNCS 3443.
14. A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps, "Directed proof generation for machine code," in *22nd Computer Aided Verification (CAV 2010)*, pp. 288–305, 2010. LNCS 6174.
15. J. Kinder, F. Zuleger, and H. Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 214–228, 2009. LNCS 5403.
16. J. Kinder and D. Kravchenko, "Alternating control flow reconstruction," in *13th International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 267–282, 2012. LNCS 7148.
17. S. Nanda, W. Li, L. Lam, and T. Chiueh, "BIRD: Binary interpretation using runtime disassembly," in *4th Code Generation and Optimization (CGO 2006)*, pp. 358–370, 2006.
18. M. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Recurring Malcode 2007*, pp. 46–53, 2007.
19. T.Izumida, K.Futatsugi, and A.Mori, "A generic binary analysis method for malware," in *5th International Workshop on Security*, pp. 199–216, 2010. LNCS 6434.

20. S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The bincoa framework for binary code analysis," in *23rd Computer Aided Verification (CAV 2011)*, pp. 165–170, 2011. LNCS 6806.
21. J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," in *Symposium on Requirements Engineering for Information Security (SREIS'01)*, 2001.
22. P. Singh and A. Lakhotia, "Static verication of worm and virus behavior in binary executables using model checking," in *4th IEEE Information Assurance Workshop*, 2003.
23. F. Song and T. Touili, "Efficient malware detection using model-checking," in *FM 2012: Formal Methods*, pp. 418–433, 2012. LNCS 7436.
24. S. Bardin, P. Baufreton, N. Cornuet, P. Herrmann, and S. Labbe, "Binary-level testing of embedded programs," in *Proceedings of the 13th International Conference on Quality Software (QSIC 2013)*, pp. 11–20, 2013. LNCS 4414.
25. S. Bardin and P. Herrmann, "OSMOSE: automatic structural testing of executables," *International Journal of Software Testing, Verification and Reliability (STVR)*, pp. 29–54, 2011.
26. P. Royal, M. alpin, D. Dagon, and R. Edmonds, "Polyunpack: Automating the hidden code extraction of unpack-executing malware," in *Computer Security Applications Conference 2006 (ACSAC06)*, pp. 289–300, 2006.
27. G. Bonfante, J.-Y. Marion, and D.R.-Plantey, "A computability perspective on self-modifying programs," in *7th Software Engineering and Formal Methods (SEFM 2009)*, pp. 231–239, 2009.
28. R. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *European software engineering conference*, vol. 30(5), pp. 263–272, 2005.
29. N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *5th European Dependable Computing Conference*, pp. 281–292, 2005. LNCS 3463.
30. K. Sen and G. Agha, "Cute and jcute : Concolic unit testing and explicit path model-checking tools," in *18th Computer Aided Verification (CAV 2006)*, pp. 419–423, 2006. LNCS 4414.
31. P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: Whitebox fuzzing for security testing," in *ACM Queue*, pp. 40–44, 2012.
32. J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Programming Language Design and Implementation (PLDI 2004)*, pp. 131–144, 2004.
33. M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa, "A hybrid approach for control flow graph construction from binary code," in *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC 2013)*, pp. 159–164, 2013.
34. T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," in *Sci. Comput. Program.*, vol. 58, pp. 206–263, 2005.