Obfuscation code localization based on CFG generation of malware

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

[1]Ho Chi Minh City University of Technology, Vietnam
[2]Japan Advanced Institute of Science and Technology, Japan

Abstract. This paper presents a tool BE-PUM (Binary Emulator for PUshdown Model generation), which generates a precise control flow graph (CFG), under presence of typical obfuscation techniques of malware, e.g., *indirect jump*, *self-modification*, *overlapping instructions*, and *structured exception handler (SEH)*, which cover *packers*. Experiments are performed on 2000 real-world malware examples taken from *VX Heaven* and compare the results of a popular commercial disassembler IDA Pro, a state-of-the-art tool JakStab, and BE-PUM. It shows that BE-PUM correctly traces CFGs, whereas IDA Pro and JakStab fail. By manual inspection on 300 malware examples, we also observe that the starts of these failures exactly locate the entries of obfuscation code.

Keywords: concolic testing, binary code analysis, malware, obfuscation

1 Introduction

Recently, control flow graphs (CFGs) attract attention in malware detection at industry levels, e.g., VxClass at Google. They use semantic fingerprints [24,29,36] to overcome the limitation on advanced polymorphic viruses of bits-based fingerprints [35,15], which are popular in commercial antivirus software. Semantic fingerprints consist of code and control flow graph fragments, which are obtained by disassembly. Then, the similarity is detected by statistical methods.

Beyond detection, malware classification requires more precise control flow graphs to observe what kinds of obfuscation/infection techniques are used. However, precise disassembly is not easy. For instance, commercial disassemblers, e.g., IDA Pro and Capstone, are easily cheated by typical obfuscation techniques, like *indirect jump*, *structured exception handler*, *overlapping instructions*, and *self-modification*. Typical self-modification is a self-decryption, and often by modifying encryption keys, a polymorphic virus mutates. Worse, recent *packers*, e.g., UPX, Themida, Telock, PECompact, Yoda, give us easy generation of polymorphic viruses.

Our ultimate goal is malware classification by their obfuscation techniques. As the first step, the aim of this research is to generate a precise CFG of x86/Win32 binary under presence of typical obfuscation techniques, which includes precise disassembly. As a byproduct, we observe that when the results of IDA Pro and BE-PUM differ, they locate the entries of the obfuscation code.

Corresponding to the nature of dynamic parsing of x86, we apply on-the-fly CFG generation. To handle self-modification, we regard a CFG node as a pair of a location and an instruction, such that a modified code is regarded as a different node. They are also applied in McVeto [37], but different from its CEGAR approach, we apply dynamic symbolic execution (concolic testing) in a breadth-first manner to decide the next destinations of multiple paths at a conditional jump [12]. Concolic testing requires a binary emulator. Our choice is to restrict the binary emulation to a user process, and APIs are handled by stubs. This gives the flexibility to handle anti-debugging and trigger-based behavior [9], at the cost of manual stub construction and approximation.

The framework is implemented as BE-PUM (Binary Emulator for PUshdown Model generation), and experiments are performed on 2000 real-world malware taken from VX Heaven[1] to compare the results of a popular commercial disassembler IDA Pro, a state-of-the-art tool JakStab, and BE-PUM. It shows that BE-PUM correctly traces CFGs, whereas IDA Pro and JakStab fail. By manual inspection on 300 malware examples, we also observe that the starts of the failures exactly locate the entries of obfuscation code.

Contributions Each element of the techniques in BE-PUM is not new, e.g., on-the-fly control flow graph generation [37,38], dynamic symbolic execution (concolic testing) [12,13], and formal x86 (32bit) semantics [7]. Dynamic symbolic execution for precise CFG generation is also not new, e.g., for C [34] and x86 binaries of system software [28]. Our contributions are:

- We compose them as a tool BE-PUM to generate precise CFG of x86 binary under the presence of obfuscation techniques, and its precision and practical efficiency are confirmed by empirical study.
- A preliminary BE-PUM was presented in [27], which supports 18 x86 instructions, no Windows APIs, and no self-modifying code. Current BE-PUM is extended to support about 200 x86 instructions and 310 Win32 APIs.
- We observe that when the results of IDA Pro and BE-PUM differ, they correctly locate the entry points of obfuscation codes.

CFGs generated by disassembler tools can be used to build models for model checking malware [4,32,33,16,18,19]. For instance, a CFG generated by IDA Pro was used for this purpose in [32,33]. Our approach immediately boosts such model checking by providing more precise models.

Another popular method to detect malware behavior is dynamic execution on binary emulators. However, malware behavior observation is sometimes not enough for classifying techniques. Even worse, dynamic execution may miss hidden behavior of malware. For instance, malware detects that they are in a sandbox by anti-debugging techniques, e.g., observing response time, checking behavior of rarely used Windows APIs, and calling the API "IsDebuggerPresent". Another difficulty is trigger-based behaviors [9,31,25], e.g., attacks triggered by specific date and time. We believe that semantic understanding of malware will compensate these limitations.

[1] `http://vx.netlux.org`

2 Typical obfuscation techniques and their difficulty

Roughly speaking, malware techniques consist of three steps.

1. Obfuscation, e.g., complex control flow to get rid of the bit-based detection, and anti-debugging to hide malicious intention during sandbox emulation.
2. Infection / spreading techniques, e.g., attacking Windows security holes.
3. Malicious behavior, e.g., information leak.

We focus on the first step. Nowadays, at least 75% of malware uses a packer [30]. The original aim of a packer is the code compaction, and later includes obfuscation techniques to evade reverse engineering for software license protection. Most of control flow obfuscation are combinations of techniques below.

- **Indirect jump.** This technique stores the target of the jump in a register, a memory address, or a stack frame (jumping with `ret`), of which these values are often modified with arithmetic operations. It also appears as *overlapping instruction* [21], which confuses the boundary of instructions at the binary level.
- **Structured exception handler (SEH).** When exceptions like *division by zero* and *write on protected area* occur, the control is spawn to a system error handler and the stack is switched to another memory area in the user process. When the system error handler ends, the control returns to the user process, and the stack is recovered to the original. An SEH is an exception handler in a user process, prepared for post processing of an exception. SEH techniques often modify the return address at `fs:[0x00]`.
- **Self modifying code (SMC).** During the execution, binary code loaded on memory is modified. Often, it appears as *Self decryption*, in which the execution of a header part modifies the later part of binary code.
- **Entry point obscuring.** The entry point is set to outside the .code section.

IDA Pro is the most popular commercial disassembler, which combines recursive disassembly and linear sweep. We show obfuscation examples, in which IDA Pro is confused.

Indirect jump

Indirect jump hides the control flow by storing the target of a jump in a register or memory. *Virus.Adson.1559* shows a typical approach to dynamically load a library by calling Windows API *GetProcAddress* for retrieving the address of the target API. By calling convention of the API, the return value of *GetProcAddress* is stored in the register *eax* and it calls the API by jumping to the value of *eax*.

```
004024A6   50              PUSH EAX ; FindFirstFileA
004024A7   FFB5 36324000   PUSH DWORD PTR SS:[EBP+403236]
                                ; Kernel32 Handle
004024AD   FF95 3A324000   CALL DWORD PTR SS:[EBP+40323A]
                                ; Call GetProcAddress
004024B3   FFE0            JMP EAX ; Call FindFirstFileA
```

IDA Pro fails to resolve the next address at 004024B3, and JakStab as well.

Overlapping instruction

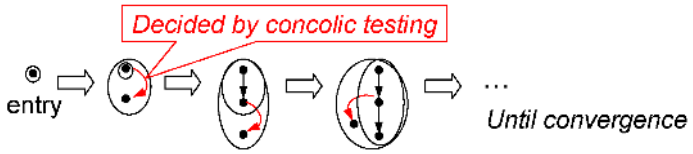
From $0040B332$ to $0040B334$ and $0040B326$ to $0040B32C$ in *Virus.Bagle.bf* (from the VX Heaven), it pops the value $0040B08F$ from the stack (as *esp* points to $0012FFA4$) to *ecx* and increments its value. Then, it pushes the value $0040B090$ of *ecx* to the stack again. At $0040B32C$, `ret` jumps to $0040B090$, where the binary code is `EB 13 EB 02` and interpreted as `jmp 0x0040b0a5`, whereas IDA-Pro jumps to $0040B08F$ and fails to interpret `E8 EB 13 EB 02`.

```
                              IDA-Pro          Correct CFG
0040B08A E89E020000 call sub_40B32D
0040B08F E8EB13EB02 call near ptr 32BC47Fh
0040B090   EB13FB02                          jmp 0x0040b0a5
...
0040B326 EB03       jmp short loc_40B32B     jmp 0x0040b32b
0040B32B 51         push ecx                 pushl %ecx
0040B32C C3         retn                     ret
...
0040B332 59         pop ecx                  popl  %ecx
0040B333 41         inc ecx                  incl  %ecx
0040B334 EBF0       jmp short loc_40B326     jmp 0x0040b326
```

3 CFG reconstruction techniques

3.1 Concrete model and its on-the-fly generation

The execution of binary code is dynamic, i.e., interpret a binary sequence that starts from a memory address pointed by the register `eip`, which decides the next address to set `eip`. Corresponding to such nature, our CFG construction is designed in an on-the-fly manner. In the figure below, when a CFG node is a conditional jump, we apply concolic testing to decide next destinations.



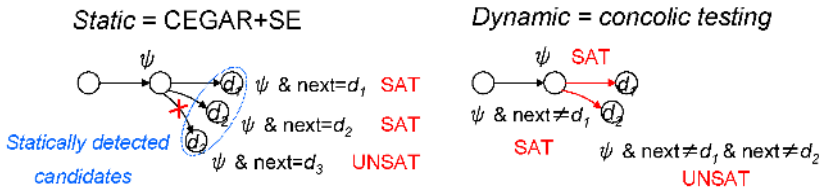
The state of a binary program can be regarded as an environment consisting of values of registers, flags, and a memory status (which includes the status of the stack). Our concrete model represents such a state by a pair $\langle (k, asm), \psi(\bar{\alpha}) \rangle$ of a CFG node and a path condition $\psi(\bar{\alpha})$ of a path $\bar{\alpha}$ reaching from the initial CFG node to (k, asm). This path condition encodes the environment after the execution from the entry point. We adopt a pair of a location and an instruction as a CFG node to handle self-modifying code. When self-modification occurs, we distinguish (k, asm) and (k, asm') as different CFG nodes. This idea is also used in McVeto [37]. We fix the notation as.

- k is an address in M and k_0 is the entry address,
- asm is an x86 assembly instruction,
- asm obtained by disassembly of a binary sequence starting from $k \in M$ is referred by $asm = instr(Env_M, k)$,
- $(m, asm') = next(k, asm)$ with $k = Env_R(eip)$ and $asm = instr(Env_M, k)$ is decided by a transition $Env \rightarrow Env'$ (described in Fig. 1) as $m = Env'_R(eip)$ and $asm' = instr(Env'_M, m)$.

3.2 Concolic testing and multiple path detection

Symbolic execution [22] is a traditional technique to symbolically execute a program, which maintains a symbolic state $\langle p, \psi \rangle$ where p is a CFG node and ψ is a path formula of the path from the initial CFG node to p. A path formula ψ describes the precondition of the execution path to p, starting from the precondition at the program entry. If ψ is satisfiable (often checked by SAT/SMT solvers), the path is *feasible*.

In binary code, for data instructions (e.g., MOV, ADD, XOR), the next location is statically decided by the length of the instruction. However, for control instructions (e.g., JMP, CMP), it may be dynamically decided, especially at indirect (conditional) jumps. Using symbolic execution, there are two ways to explore possibly multiple destinations of a CFG node (in the figure below).



- **Static symbolic execution (SSE)**, in which next destination candidates are statically detected, and the feasibility of each destination p' is checked by the satisfiability of $\psi \wedge \texttt{next} = p'$. To refine statically detected candidates, it can be combined with CEGAR, like in McVeto [37].
- **Dynamic symbolic execution (DSE)**, in which the feasibility is checked by testing with a satisfiable instance of ψ (*concolic testing*), which requires a binary emulator. This will continue until $\psi \wedge \texttt{next} = p' \wedge \texttt{next} = p''$.... becomes UNSAT for explored next destinations $p', p'', ...$, like in [12].

We describe a path formula as a pair of an environment of parameters (e.g., registers, flags) and a Presburger formula consisting of constants and symbolic values of inputs, which is obtained by deploying the current values of parameters as the updates of their initial values. It ignores system status and kernel procedures, and focuses only on a user process.

3.3 API as stub

Concolic testing requires a binary emulator, and our choice is to restrict a binary emulation to a user process, and APIs are handled by stubs. This gives the flexibility in symbolic execution, at the cost of manual stub construction and an approximation. Current BE-PUM implements the stubs for 310 APIs. The output of an API in the stub is given either by Java API or as a symbolic value. Note that a stub keeps a post-condition as the same as the pre-condition, but updates the environment by its output.

- For typical APIs functions, such that *FindFirstFileA, GetModuleHandle, FindNextFileA*, we rely on JNA (Java Native Access), which allows Java programs to call native shared libraries. Each system call is treated as if a single instruction such that (i) the update of the path formulas and the environments follow to the technical specification at Microsoft Developer Network.3; and (ii) the return value is obtained by executing Java API. Note that most of major APIs (though not all APIs) are covered by Java API.
- Sometimes, specific APIs are used for obfuscation techniques, e.g., anti-debugger or anti-debugging, and trigger-based behavior. For them, we must avoid execution of the API, since it will make us fall into the "trap" of the malware. For example, some malware performs anti-emulator by calling the system call like *IsDebuggerPresent* or *CheckRemoteDebuggerPresent*. If this API is executed under an emulator, a specific value will be returned, and the malware changes its behavior to hide its intention. For them, instead of calling Java API, we simply adopt a symbolic value as an output of the API, like in [9]. This is also effective for trigger-based behavior.

4 Control flow graph reconstruction

4.1 X86 operational semantics

Our x86 binary semantics are inspired by [7]. We assume that a target X86 binary program $Prog_{x86}$ is loaded on a memory area, referred as M. The instruction pointer *eip* and the stack pointer *esp* are special registers that point to the current address of instructions and the top of the stack, respectively. The former is initially set to the entry address of $Prog_{x86}$. The stack is taken in M, where the stack top frame is pointed by the register *esp* and the stack bottom is pointed by the register *ebp*. In Windows, the stack area is taken between the *stack base* and the *stack limit*, which has the length of 1M bytes (and can be enlarged); but we ignore these boundaries.

Definition 1. *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).

For $k = Env_R(eip) \in M$, let $instr(Env_M, k)$ be a mapping that disassembles a binary code at the memory location k and return an instruction (with its arguments). An operational semantics *of binary codes $Prog_{x86}$ is described as transitions in Fig. 1 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 $M \setminus S$).*

In BE-PUM, each register in R is represented by a 32-bit vector. Meanwhile, system flags in F are simply represented as boolean variables. Each memory location in M is represented by a 8-bit vector, in which the arithmetic operations are bit-encoded. BE-PUM dynamically identifies the instruction boundary as a sequence of 8-bit vectors (instead of a fixed 32-bit segment in the preliminary version [27]), which helps us to handle *overlapping instructions*.

$$\frac{\begin{array}{c} Env_R(eip) = k, instr(Env_M, k) ='' \ call \ r'', \\ m' = k + |call \ r|, m = Env_R(r), push(S, m') = S' \end{array}}{(Env_F, Env_R, Env_S, Env_M) \rightarrow (Env_F, Env_R[eip \leftarrow m, \ esp \leftarrow esp - 4], Env_{S'}, Env_M)} \ [Call]$$

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

$$\frac{Env_R(eip) = k, instr(Env_M, k) ='' \ ret'', \neg empty(S), pop(S) = (S', m)}{(Env_F, Env_R, Env_S, Env_M) \rightarrow (Env_F, Env_R[eip \leftarrow m, \ esp \leftarrow esp + 4], Env_{S'}, Env_M)} \ [Return]$$

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

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

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

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

Fig. 1: Some of rules of operational semantics for control instructions

Fig. 1 shows some examples of the description of the operational semantics of x86 instructions, which follows the technical description at *Intel Software*

Developer's Manual. For instance, in the instruction $Call$, the register *eip* points to the address k in memory M of the next instruction, and $instr(Env_M, k)$ maps the binary code at k to the next instruction *call* r. The return address of *call* r is calculated by adding the size of the instruction *call* r to the current address k and is pushed onto the top of the stack S. The address of next instruction is updated with the value m of memory pointed by the address r.

4.2 Concrete model with path conditions and CFG reconstruction

Definition 2. *We borrow notations from Definition 1. A control flow graph (CFG) node is a pair of a location k and and an assembly instruction asm. A configuration of a concrete model is a pair $\langle (k, asm), \psi(\bar{\alpha})\rangle$ where (k, asm) is a CFG node, $\bar{\alpha}$ is a path (a sequence of CFG nodes) from the initial CFG node (k_0, asm_0) to (k, asm), and $\psi(\bar{\alpha})$ is a path condition given by*

$$\begin{cases} \psi(\epsilon) := \textbf{true} \\ \psi(\bar{\alpha}') := \psi(\bar{\alpha}) \wedge (SideCond \wedge PostCond) \text{ if } \bar{\alpha}' = \bar{\alpha}.next(k, asm) \end{cases}$$

for the side conditions $SideCond$ appearing in $Env \rightarrow Env'$ and the strongest post condition $PostCond$ at (k, asm), which is a Presburger formula consisting of constants and symbolic values of inputs.

We obtain a CFG of an x86 binary program $Prog_{x86}$ by extracting (k, asm) from a configuration $\langle (k, asm), \psi(\bar{\alpha})\rangle$ in Definition 2. There are several reasons for causing branching on a CFG. $next(k, asm)$ may have multiple possibility depending on an initial environment, which may be given by external system status. In current BE-PUM implementation, possible $next(k, asm)$'s are explored by repeated concolic testing with its satisfiable instances. This exploration continues until it reaches to UNSAT by adding the refutations of already explored next destinations.

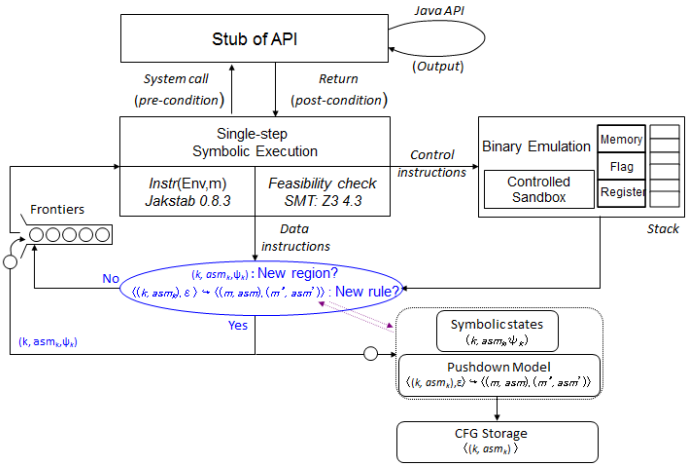
5 BE-PUM implementation

5.1 BE-PUM architecture

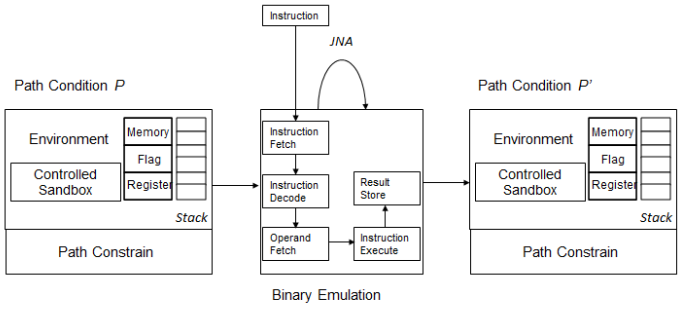
BE-PUM implements CFG reconstruction (in Definition 2) based on concolic testing. It applies *JakStab 0.8.3* [20] as a preprocessor to compute a single-step disassembly $instr(Env_M, k)$, and an SMT *Z3.4.3* as a backend engine to generate a test instance for concolic testing.

The figure below shows the architecture of BE-PUM, which consists of three components: *symbolic execution*, *binary emulation*, and *CFG storage*. The symbolic execution picks up one from the frontiers (symbolic states at the ends of explored execution paths), and it tries to extend one step. If the instruction is a data instruction (i.e., only Env_M is updated and the next location is statically decided), it will simply disassemble the next instruction. If the instruction is a control instruction (e.g., conditional jumps), the concolic testing is applied to

decide the next location. Note that some variable does not appear in the path-condition, the SMT will not return its value. If the concolic testing needs this value, BE-PUM terminates. However, in our observation, this is unlikely. When either a new CFG node or a new CFG edge is found, they are stored in CFG storage and a configuration is added to the frontiers. This procedure continues until either the exploration has converged (which will be discussed later), or reaching to unknown instructions, system calls, and/or addresses.



The next figure shows the implementation of a stub in BE-PUM, which consists of three components: a *pre-condition* P, the *binary emulation*, and a *post condition* P'. Currently, the BE-PUM implementation passes P to P' as identical, but with the update of the environment by an output of an API, which is obtained together with the return address by executing native shared library in JNA.



Note that exceptions, like the *division by zero*, are detected at the binary emulator, which pass them to the Windows system error handler.

5.2 Strategy and limitations

There are several limitations in our implementation.

- X86 instructions are about 1000 and Windows APIs are more than 4000. Current BE-PUM covers only 200 x86 instructions and 310 APIs. They are selected by the frequency appearing in malware from VX Heaven.[2] Frequency is initially estimated by JakStab, and bootstrapped by observing unexpected termination of BE-PUM by *unknown instructions* or *unknown API*.
- Since CFG reconstruction is by a bounded symbolic execution, complete CFG reconstruction requires *loop invariant generation*. Currently, BE-PUM simply unfolds loops and if the same CFG node is visited one million times, it terminates. This naive choice still works, since most of loops in malware have the fixed number of iterations for the self-decryption. Another reason is, due to the limited support of x86 instructions and Windows API, BE-PUM terminates more by unknown instructions and APIs.
- The binary emulation in BE-PUM limits its scope only on a user process, and handling APIs by manually prepared stubs. This gives the flexibility to handle anti-debugger and trigger-based behavior [9], at the cost of manual stub construction and approximation.

BE-PUM also adopts choices on initial setting.

- When BE-PUM starts to generate a model of binary code, the stack S is initially pushed (i) the address of the file name of the code; (ii) the address of the system exception handler; and (iii) the return address, which is randomly selected from the memory image of *kernel32*. The former two obey to the standard manner of Win32, and the last is based on the assumption that a code is often started by a call from somewhere in *kernel32*. This frequently holds and is often used by malware, e.g., *Aztec* [23].
- The initial environment (e.g., registers and flags) follows to the reference[3].
  - **Flags**: CF = **False**, PF = **True**, AF = **False**, ZF = **True**, SF = **False**, TF = **False**, DF = **False**, OF = **False**, IF = **False**.
  - **Registers**: $EAX, CS, DS, ES, FS, GS, SS, EFFLAGS$ are set to symbolic values, EIP and EDX to the address of the entry point, ESP and EBP to the addresses of the top and the base of the stack, respectively. The rest is set as $ECX = 0, EBX = 7EFDE00, EDI = 0, ESI = 0$.

6 Experiments

We perform experiments of CFG reconstruction for 2000 real-world malware, taken from *VX Heaven*, and 6 non-malware examples. We compare the results of BE-PUM with IDA Pro and JakStab for the coverage of nodes, edges, and processing time. Our experiments are performed on Windows XP with AMD Athlon II X4 635, 2.9 GHz and 8GB. Note that BE-PUM may have stopped with an *unknown instruction* or *API*.

[2] `http://vx.netlux.org`
[3] `https://code.google.com/p/corkami/wiki/InitialValues`

6.1 Model generation performance

In general, the number of reachable nodes of BE-PUM is better than IDA Pro and JakStab. Since IDA Pro uses syntactic analysis, it is easily confused by indirect jumps and encrypted codes. JakStab also fails since its static analysis cannot effectively analyze. BE-PUM encounters the problem at the cost of higher processing time than JakStab and IDA Pro due to its concolic testing.

As a statistical observation of the experimental result, we show the averages of the ratios among tools.

- The ratio between the numbers of CFG edges and CFG nodes, which shows detection of multiple paths. The ratios for JakStab, IDA Pro, and BE-PUM are 1.05, 2.05, and 1.24, respectively. This shows that IDA Pro detects multiple paths better, but they could be infeasible paths.
- The ratio between the numbers of explored CFG nodes by two tools. Between BE-PUM and JakStab is 11.59, and between BE-PUM and IDA Pro is 6.61. Notable examples in Table 1 are, *Virus.Artelad.2173, Email-Worm.LoveLetter.b, Virus.Pulkfer.a*, and *Email-Worm.Klez.h*, in which nodes are about 10 times more by BE-PUM. Since the numbers of CFG nodes and edges by BE-PUM are almost the same, BE-PUM fails to find multiple paths. This can occur when the loop counter is set to a constant and the loop continues to decrypt/modify some fragment. Further investigation is needed on these examples.

With closer look, Table 1 shows 28 malware examples among 2000 and 6 non-malware examples, in which the unit of **Time** is the millisecond. Among 2000, BE-PUM successfully converges with 250 examples, and it is interrupted on many viruses by *unknown instructions* (e.g., *Email-Worm.Bagle*) and *unsupported APIs* (e.g., *Virus.Cabanas*) due to the limited support of instructions and APIs. It is also interrupted by *unknown addresses* (e.g., *Virus.Seppuku* except for *Seppuku.1606*) due to jumping to an address outside the file area (e.g. the address of an unknown API or a system file).

IDA Pro sometimes detects more nodes. It is partially because IDA Pro covers most of x86 instructions, whereas BE-PUM covers only 200. Another reason is that IDA Pro is cheated by obfuscation and continues to generate unrelated assembly code, e.g., *Benny.3219.a/b* and *Eva.a/b*, whereas BE-PUM terminates with a precise CFG. This also occurs in non-malware examples. IDA Pro often has better results than BE-PUM because of its full support of Windows APIs. For example, with *Winever.exe*, BE-PUM fails to resolve the return address of Windows API *ShellAboutW@shell32.dll*.

```
01001284  56              PUSH ESI
01001285  56              PUSH ESI
01001286  8D45 BC         LEA EAX,DWORD PTR SS:[EBP-44]
01001289  50              PUSH EAX
0100128A  56              PUSH ESI
0100128B  FF15 3C100001   CALL ShellAboutW@Shell32.dll
```

However, in *systray.exe*, BE-PUM is better than IDA Pro, since IDA Pro fails to resolve obfuscation techniques, like an indirect return at

```
010010BD   C2 0400          RETN 4
```

and an indirect jump at

```
010010CA   6A 00            PUSH 0
010010CC   BF 5C100001      MOV EDI, 0100105C
010010D1   57               PUSH EDI
010010D2   FFD6             CALL ESI
```

Table 1: Part of experimental results for model generation

Example	Size	JakStab			IDA Pro			BE-PUM		
	KByte	Nodes	Edges	Time	Nodes	Edges	Time	Nodes	Edges	Time
Virus.Artelad.2173	23	134	154	10ms	159	162	1133ms	1610	1611	236468ms
Email-Worm.LoveLetter.b	60	1027	1026	297	984	1011	10	7558	7602	1073984
Virus.Pulkfer.a	129	907	924	10	805	823	20	8347	8353	44672
Email-Worm.Klez.h	137	192	178	20	50	56	1	5652	5651	46344
Email-Worm.Coronex.a	12	26	27	500	148	157	204	308	339	1000
Trojan-PSW.QQRob.16.d	25	89	100	766	17	15	382	91	105	953
Virus.Aztec	8	104	111	1973	223	215	495	300	313	44384
Virus.Belial.a	4	41	42	407	118	116	198	128	134	985
Virus.Benny.3219.a	8	138	153	890	599	603	415	149	164	2438
Virus.Benny.3223	12	42	47	328	770	781	135	149	164	2218
Virus.Bogus.4096	38	87	98	546	88	86	269	88	98	656
Virus.Brof.a	8	17	17	343	98	102	167	137	147	1484
Virus.Cerebrus.1482	8	6	5	156	164	165	70	179	198	735
Virus.Compan.a	8	25	26	360	83	81	176	91	98	484
Virus.Cornad	4	21	20	141	68	72	67	94	100	344
Virus.Eva.a	8	14	13	329	381	392	145	249	277	13438
Virus.Htrip.a	8	10	10	359	145	143	172	148	157	2187
Virus.Htrip.d	8	10	10	265	164	162	124	165	173	2296
Virus.Seppuku.1606	8	131	136	1968	381	390	965	339	364	8372
Virus.Wit.a	4	54	60	360	153	151	172	185	203	2641
Email-Worm.Bagle.af	21	123	143	937	142	151	461	140	166	2157
Email-Worm.Bagle.ag	17	127	147	828	13	12	413	127	147	1047
Virus.Cabanas.a	8	3	2	156	1	1	78	68	72	1532
Virus.Cabanas.b	8	3	2	140	9	7	70	63	66	1781
Virus.Canabas.2999	8	2	1	656	7	6	85	358	401	8703
Virus.Seppuku.1638	8	139	144	2266	414	412	112	689	712	13000
Virus.Seppuku.3291	8	26	25	187	556	554	66	253	270	12156
Virus.Seppuku.3426	8	27	27	188	30	28	61	299	317	13484
non-malware binary										
hostname.exe	8	329	360	2412	343	389	33	326	357	235610
winver.exe	6	162	166	422	310	345	24	232	240	122484
systray.exe	4	110	136	532	115	138	14	123	139	16125
regedt32.exe	3	52	54	266	56	61	11	61	69	22844
actmovie.exe	4	164	179	281	187	215	51	180	196	243469
nddeapir.exe	4	164	179	500	187	215	24	180	196	223297

6.2 Example of obfuscation localization and classification

We show a case-study of *Seppuku.1606*. Most of malware in VX Heaven are traditional and some have known assembly source (e.g., for *Aztec*, *Bagle*, *Benny*, and *Cabanas*), where *Seppuku.1606* has no available assembly source. We manually traced the result of *BE-PUM* with the help of *Ollydbg*, and found code fragments for SEH technique and self-modification.

An intended exception occurs at 00401035 by reading the memory address $77E80000$ pointed by register *esi*, and then storing that value in the register `eax`. Since $77E80000$ is protected, this raises an exception caught by an SEH.

```
00401028 33C0           xor   eax, eax
0040102A 64FF30         push  dword ptr fs:[eax]
0040102D 648920         mov   fs:[eax], esp
00401030 BE0000E877     mov   esi, 77E80000h
00401035 66AD           lods  ds:[esi]
```

The self-modification occurs at $004010EB$ that *Seppuku.1606* overwrites the opcode at 00401646 from $E8FFFFF9B5$ to $E800000000$, which means the modification from $Call$ 00401000 to $Call$ $0040164B$.

```
004010E4 57             PUSH EDI
004010E5 8B8589144000   MOV EAX,DWORD PTR SS:[EBP+401489]
004010EB AB             STOS DWORD PTR ES:[EDI]
004010EC 83C404         ADD ESP,4
```

In the correct CFG, the path from $0040164B$ finally reaches the exit point after a call of an API *MessageBoxA*. BE-PUM correctly traces this behavior, where IDA Pro fails at 00401646 with `call sub_401000`.

```
00401000 60             PUSHA
00401001 E800000000     CALL $+5
.....
00401646 E800000000     CALL Virus_Wi.0040164B
0040164B 6A10           PUSH 10
0040164D 6800204000     PUSH Virus_Wi.00402000
00401652 6827204000     PUSH Virus_Wi.00402027
00401657 6A00           PUSH 0
00401659 E80D000000     CALL <JMP.&USER32.MessageBoxA>
0040165E 6A00           PUSH 0
00401660 E800000000     CALL <JMP.&KERNEL32.ExitProcess>
```

The CFG generated by IDA Pro has a wrong path at the instruction $004010EB$, whereas BE-PUM successfully continued as this point. We observe that $004010EB$ is in fact the entry point of the self-modification. A similar situation occurs for all of other 285 obfuscation samples (Section refsec:ManualInsp).

6.3 Manual obfuscation classification

We expect that when the results of IDA Pro and BE-PUM differ, they will locate the entry points of obfuscation code. To confirm this idea, we choose 300 viruses

among 2000 from VX Heaven, and generate CFGs by BE-PUM and IDA Pro. They are automatically compared, and the former is manually investigated at the point that the difference occurs with the help of Ollydbg. We observe that 293 viruses contain obfuscation code and they are classified into

- *249 indirect jumps*: Virus.Delf.n, Virus.Delf.r, Virus.Delf.w, Worm.Randin.b, Worm.Limar, Worm.Delf.q, Worm.Delf.o, ...
- *110 SEH*: Virus.Eva.a Virus.Eva.b, Virus.Eva.c, Virus.Eva.e, Virus.Eva.f, Virus.Eva.g, Virus.Rever, ...
- *30 self-modifying code*: Virus.Cabanas.2999 Virus.Rever, Net-Worm.Sasser.b, Net-Worm.Sasser.c, Virus.Pesin.a, ...
- *6 encryption:* Virus.Cabanas.2999, Virus.Savior.1828, Net-Worm.Sasser.a, Net-Worm.Sasser.f, Virus.Glyn, Virus.Hader.2701.

and all of the cases, when the results of IDA Pro and BE-PUM differ, they exactly locate where obfuscation starts. There are 7 viruses without obfuscation code, and IDA Pro and BE-PUM report the same result.

As an example of indirect jumps (by `retn`, similar to Virus.Bagle.bf in Section 2), we observe Virus.HLLW.Rolog.f.

```
                          BE-PUM          IDA Pro
00437001 60               PUSHAD
00437002 E8 03000000      CALL 0043700A
00437007 E9 EB045D45                    JMP 45A074F7
00437008    EB04          JMP 0043700E
0043700A 5D               POP EBP
0043700B 45               INC EBP
0043700C 55               PUSH EBP
0043700D C3               RETN
```

At 00437002, the Call instruction put the return address 00437007 into the stack. From 0043700A to 0043700D, 00437007 in the top stack frame is popped to the register *ebp*, incremented, and pushed again. Thus, at 0043700D, it must return to 00437008, instead of 00437007.

BE-PUM correctly read `JMP 0043700E`, whereas IDA Pro is confused as `JMP 45A074F7`. JakStab also handles this obfuscation.

7 Related Work

There are two main targets of binary code analysis. The first one is *system software*, which is compiled code but its source is inaccessible, due to legacy software and/or commercial protection. It is often large, but relatively structured from the compiled nature. The second one is *malware*, which is distributed in binary only. It is often small, but with tricky obfuscation code. For the former, a main obstruction is scalability. *IDA Pro* [4] and *Capstone* [5] are the most pop-

[4] `https://www.hex-rays.com/products/ida/`
[5] `http://www.capstone-engine.org/`

ular disassemblers. Other remarkable approaches include *OSMOSE* [5] built on *BINCOA* [6], *BitBlaze* [13], *Veritesting* [1] built on *BAP* [10], and *SAGE* [28].

There are various model generation tools for binary executables. Destinations of indirect jumps can be identified by either static or dynamic method. BE-PUM stands in between, using both methods. For instance, CodeSurfer/x86 [2], McVeto [37], and JakStab [20]) adopt a static approach, while *OSMOSE* [5], BIRD [26], Renovo [17], Syman [38], and SAGE [28] choose a dynamic one. Generally, dynamic methods seems more effective on malware analysis [11].

Static methods are abstract interpretation (static analysis) for an over approximation, and symbolic execution to check feasibility of an execution path for an under-approximation. JakStab and McVeto apply CEGAR to refine over approximations. McVeto also uses symbolic execution to check the path feasibility (named a *concrete trace*). X-Force [14] executes dynamically, which is implemented on PIN from Intel. It explores different execution paths by systematically forcing the branch outcome of conditional transfer instructions. BE-PUM uses dynamic symbolic execution, instead of dynamic execution. BitBlaze [13] also combine static and dynamic analyses on binary. It is based on Value Set Analysis (VSA) [3] for handling indirect jumps. It also adopts existing disassemblers like IDA Pro, and handling obfuscated code relies on them. Moser [12] developed a system for Windows to explore multiple paths. In [12], it is mentioned not to cover self-modification. MineSweeper [9] uses symbolic values for APIs to detect trigger-based behavior. In [9], it is mentioned not to cover indirect jumps. Syman [38] emulates the full Windows OS. Such detailed information makes symbolic execution complex and models easily explode even for small binary code. OSMOSE and CodeSurfer/x86 reduce to 32-bit vector models, called DBA (Dynamic Bit-vector Automaton) [5]. *CoDisasm* [8] is proposed very recently, which disassembles based on analyses of dynamic traces by decomposing into *waves*. It also can handle overlapping instructions and self-modification, and further comparison is needed.

In summary, BE-PUM is the most similar to McVeto. However, McVeto finds candidates of the destinations by static analysis (which are possibly infinitely many), and chooses one and checks the satisfiable condition of the path at the destination to conclude whether it is reachable (concrete trace). BE-PUM solves the path condition at the source of an indirect jump, and applies concolic testing (over binary emulator) to decide (one of) the destination. Unfortunately, we could not find access to the McVeto implementation, and we did not compare with experiments. McVeto seems not to support APIs, which limits to analyze SEH techniques. Concolic testing is easier to adopt stubs for APIs. The table below summarizes the comparison, in which ? means not confirmed yet.

	Indirect Jump	Overlapping Instruction	SEH	SMC Self-Decryption	Packer
IDA Pro	No	No	No	No	No
JakStab	Static analysis	Static	No	No	No
MineSweeper	No	No	No	No	No
X-Force	Dynamic emulation	Yes	No	Yes	?
McVeto	Symbolic execution	Yes	No	Yes	?
CoDisasm	Dynamic analysis	Yes	?	Yes	Yes
BE-PUM	Concolic testing	Yes	Yes	Yes	Yes

8 Conclusion

This paper introduced a tool BE-PUM (Binary Emulator for PUshdown Model generation), which generates a precise control flow graph (CFG) of malware under presence of typical obfuscations, based on dynamic symbolic execution on x86 binaries. Experiments are performed over 2000 malware examples taken from VX-Heaven database. Although each element of the techniques in BE-PUM is not new, the combination works in practical efficiency, and is effective such that when the results of IDA Pro and BE-PUM differ, they correctly locate the entry points of the obfuscation code. This is confirmed by manual classification of 300 examples.

A precise CFG is a backbone model for model checking. Future work includes to clarify the target properties of model checking to automatic classification of obfuscation techniques. We have some observations.

- *Indirect jump* comes together with arithmetic operations on a register or a memory address that appears as an argument of a jump instruction, or a `pop-inc-push` sequence for the overlapping instruction technique (as in *Bagle.bf* and *Heher.j*).
- *SEH* set up the return address in SEH by the specific sequence of instructions `push fs:[0]` and `mov esp, fs:[0]`.
- *SMC* comes together with a loop of XORing and a sequence in the CFG that has previously visited locations with modified instructions.

Another future work is loop handling. BE-PUM applies concolic testing to decide the destinations, but symbolic execution is a bounded search. Current BE-PUM simply unfolds loops, relying on the observation that most of loops in malware have the fixed number of iterations for self-decryption. *Loop invariant generation* is an ultimate solution; before that we are planning to apply constant propagation to detect a constant loop counter, which would improve in practice.

Acknowledgments This work is supported by JSPS KAKENHI Grant-in-Aid for Scientific Research(B) 15H02684 and AOARD-144050 (14IOA053). It is also funded by Ho Chi Minh City University of Technology under grant number TNCS-2015-KHMT-06.

References

1. T. Avgerinos, A. Rebert, S.K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *36th ICSE*, pp.1083–1094, 2014.
2. G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In *CC*, pp.250–254, 2005. LNCS 3443.
3. G Balakrishnan and T. Reps. Wysinwyx: What you see is not what you execute. *ACM TOPLAS*, 32(6):206–263, 2010. Article No.23.
4. 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 *CAV*, pp.158–163, 2005. LNCS 3576.
5. S. Bardin and P. Herrmann. OSMOSE: automatic structural testing of executables. *STVR*, pp.29–54, 2011.
6. S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA framework for binary code analysis. In *CAV*, pp.165–170, 2011. LNCS 6806.
7. G. Bonfante, J.-Y. Marion, and D.R.-Plantey. A computability perspective on self-modifying programs. In *SEFM*, pp.231–239, 2009.
8. G. Bonfante, J.Fernandez, J.-Y. Marion, B.Rouxel, F.Sabatier, and A.Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions In *CCS*, *to appear*, 2015.
9. D. Brumley, et.al. Automatically identifying trigger-based behavior in malware. In *Botnet Analysis and Defense*, pp.65–88, 2008.
10. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *CAV*, pp.463–469, 2011. LNCS 6806.
11. C.Kolbitsch, B.Livshits, B.G.Zorn, and C.Seifert. Rozzle: De-cloaking internet malware. In *IEEE sympo. Security and Privacy*, pp.443–457, 2012.
12. A Moser et al. Exploring multiple execution paths for malware analysis. In *IEEE sympo. Security and Privacy*, pp.231–245, 2007.
13. Dawn Song et al. Bitblaze: A new approach to computer security via binary analysis. In *ICISS*, 2008.
14. F. Peng et al. Force-executing binary programs for security applications. In *USENIX Security*, pp.829–844, 2014.
15. E. Filiol. Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology*, 2:35–50, 2006.
16. A. Holzer, J. Kinder, and H. Veith. Using verification technology to specify and detect malware. In *EUROCAST*, pp.497–504, 2007. LNCS 4739.
17. M. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Recurring Malcode*, pp.46–53, 2007.
18. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *DIMVA*, pp.174–187, 2005. LNCS 3548.
19. J. Kinder, et.al. Proactive detection of computer worms using model checking. *IEEE Trans. Dependable and Secure Computing*, 7:424–438, 2010.
20. J.Kinder, F.Zuleger, and H.Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI*, pp.214–228, 2009. LNCS 5403.
21. Johannes Kinder. *Static Analysis of x86 Executables*. PhD thesis, Technische Universitat Darmstadt, 2010.
22. J.C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.
23. E. Labir. VX reversing I, The basics, 2004. *manuscript*.

24. A. Lakhotia, M.D. Preda, and R. Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *PPREW*, pp.25–30, 2013.
25. A. Moser, C. Kruegel, and Andkirda. Limits of static analysis for malware detection. In *ACSAC*, pp.215–225, 2007.
26. S. Nanda, W. Li, L. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *CGO*, pp.358–370, 2006.
27. M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa. A hybrid approach for control flow graph construction from binary code. In *APSEC*, pp.159–164, 2013.
28. P.Godefroid, S.K.Lahiri, and C.R.-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, pp.112–128, 2011. LNCS 6887.
29. M.D. Preda, R. Giacobazzi, A. Lakhotia, and I. Mastroeni. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In *POPL*, pp.329–341, 2015.
30. Roundy, K. A., Miller, and B. P. Binary-code obfuscations in prevalent packer tools. In *ACM Comput. Surv*, pp.215–226, 2014.
31. M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.
32. F. Song and T. Touili. Pushdown model checking for malware detection. In *TACAS*, pp.110–125, 2012. LNCS 7214.
33. F. Song and T. Touili. LTL model-checking for malware detection. In *TACAS*, pp.416–431, 2013. LNCS 7795.
34. S.Person, M.B.Dwyer, S.G.Elbaum, and C.S.Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pp.226–237, 2008.
35. P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.
36. T.Dullien, T.Kornau, and R.-P.Weinmann. A framework for automated architecture-independent gadget search. In *WOOT*, 2009.
37. A. V. Thakur, et.al. Directed proof generation for machine code. In *CAV*, pp.288–305, 2010. LNCS 6174.
38. T.Izumida, K.Futatsugi, and A.Mori. A generic binary analysis method for malware. In *Int. Workshop on Security*, pp.199–216, 2010. LNCS 6434.