A binary code analysis and approximation techniques

By BINH, NGO Thai

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Mizuhito Ogawa

March, 2015

A binary code analysis and approximation techniques

By BINH, NGO Thai (1310022)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Mizuhito Ogawa

and approved by
Professor Mizuhito Ogawa
Professor Tachio Terauchi
Associate Professor Nao Hirokawa

February, 2015 (Submitted)

Contents

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Mizuhito Ogawa for the continuous support of not only my thesis but also my life. He taught me a lot, not only in researching, but also the attitude when we deal with a problem, and evaluating the works.

My sincere thanks also goes to Associate Professor Nao Hirokawa for his detailed and constructive comments and also for his important support through this work. I learned a lot from him how to write scientific documents.

Also I owe my loving thanks to my parents and my friends. Without their encouragement and understanding it would have been impossible for me to finish this work.

Chapter 1

Introduction

1.1 Binary code analysis

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, and the main obstruction is scalability. *IDA Pro* [1] is the most popular disassembler, combining recursive disassembly and linear sweep. The second one is *Malware*, which is distributed in binary only. It is often small, but with tricky code for obfuscation.

Many model generation tools for binary programs have been proposed [1, 9, 5, 7]). They attempt to infer a *control flow graph (CFG)* of the program, on which popular techniques like model checking can be adopted [6]. A CFG of high-level programming language is straightforward and is obtained by syntactical parsing. However, it is not easy for binary code:

Binary difficulties

– Instruction code is an element of the memory and is treated as data, which can be modified during execution. The *current state* will have the location of the current instruction in the memory, from which the instruction is parsed and executes. Without such information, there is no difference between program data and instruction code.

– Moreover, the location of the next instruction is decided by both the current instruction and the current environment. During execution, if there are unexpected operations, such as division by zero, reference to the restriction memory locations, or hardware-errors, . . . an exception will occur and control is passed to the address of the exception handler, instead of the address stated by the instruction.

For such reasons, the CFG can only be built on-the-fly. Based on Jakstab [7], a static-based binary disassembler [8], our collaborator at VNU-HCM has developed BE-PUM

[1]`https://www.hex-rays.com/products/ida/`

(Binary Emulation for PUshdown Model generation) that focuses on building the CFG of malware programs, which tend to be small in size and the code is often obfuscated.

BE-PUM generates the CFG in an under-approximation manner: it tries to find all possible execution paths and updates the graph on-the-fly. *Symbolic execution* is used to symbolically runs the program, which maintains a symbolic state (ℓ, ρ) where ℓ is a program location and ρ is the path formula of the path from program entry point to ℓ. A path formula ρ describes the precondition of the execution path to ℓ, starting from the precondition at the program entry. At one state, the location of next instruction is decided by the dynamic run on a SAT instance of the path formula. However, although we have assumed that programs are terminating, experiment data shows that BE-PUM still has the problem of infinite loop unrolling.

1.2 Problems

Consider the following program:

```
0x401000:   cmpl ecx, 0
0x401003:   jle 0x401008
0x401005:   decl ecx
0x401006:   jmp 0x401000
0x401008:   call ExitProcess
```

The program starts from the instruction at `0x401000`, and `ExitProcess` is a system API that terminates the program and passes the control back to the operating system. Here we implicitly use ecx as an input parameter. Before it executes, the system will move the input value to ecx and jump to the start instruction. Depending on such value, call n, we will go through the loop n times (for simplicity, we only decrease ecx and do nothing else within the loop). We have a conditional jump at location `0x401003`, where the location of the next instruction may be `0x401005` or `0x401008`, depending on the current value of ecx. For each path, BE-PUM will check for satisfiability of the path condition. If it is SAT, the path is viable and BE-PUM will continue the analyzing process [2]. The above program has one problem that the number of times of loop unrolling is not explicitly bounded in the code. In general, the value of ecx ranges between -2^{31} and $2^{31} - 1$. BE-PUM will blindly follow the loop without finding any new instruction.

1.3 Solution and experiments

We will apply constant propagation analysis on the partial CFG and use the analyzed data to decide whether BE-PUM will or will not continue the path. The constant propagation environment at each location maps the variables in the program (i.e. registers, memory locations, ...) to the abstract domain, which consists of three elements $\perp \sqsubseteq \mathsf{C} \sqsubseteq *$ stands for *unknown, constant* and *not-a-constant*, respectively. Let e be the constant

[2]BE-PUM v2 uses embedded emulator, and next address is calculated from the SAT instance

propagation environment at a location ℓ, and x is a variable in the program, $e(x) = \mathsf{C}$ means that regardless of the initial environment, after executing on the same path, the value of x at ℓ must be the same. As in the program, the value of ecx at any location is not-a-constant. Our approach is that if a jump revisits a node that was explored in the graph, we will continue checking the node only if the condition is a constant. This process is similar to loop unrolling, but we only unroll if there is a guarantee that the normal execution will also follow the same path, and since we have assumed that input programs are terminating, the unrolling process will also terminate.

Remark 1. The result of constant propagation is an approximation and constant propagation analysis is not sufficient to solve the infinite loop unrolling problem. There are 30% of the cases that the result CFG when applying constant propagation is smaller than the original one. Such cases can be further reduced by applying heuristic methods to detect some cases such as `xor eax, eax`, which is often used in binary code to initialize the value of a register.

We do the experiment on 1000 real-world malwares of 733 families[3] from the database of VX Heaven[4], and do the comparison between the three choices of loop unrolling:

1. (BP) BE-PUM with normal unrolling (i.e. we continue exploring the path even when it goes to an already explored location)

2. (CP) BE-PUM where we only unroll if the condition of a conditional jump is a constant. The constant propagation environment will be calculated when a new edge is added to the partial graph.

3. (CC) BE-PUM where loop unrolling is replaced by convergence check: the path is skipped if it goes to an already explored location.

Since CP is a *better* choice than CC, there is no surprise that CP always results the better graph if it terminates (907/1000 samples). Comparing CP and BP, there are 299 cases that BP finishes with better results than CP. This is expected since CP results an under-approximation of the CFG. We only unroll the loop if there is a guarantee that the path is taken for any initial environment. There have the cases that the skipped paths eventually go to new locations. We will need further analysis such as loop invariant for this problem, but it is outside of the thesis, and we regard it as future work.

Remark 2. There are cases that CP results time-out. This is because we have set the time-out limit for each run. Most of the cases involve running the fixed point operations multiple times with the big-size graph. Running CP version again without time limit on such samples eventually finishes.

[3]Each family originates from a single source base and exhibits a set of consistent behaviors
[4]`http://vxheaven.org/`

1.4 Contribution

1. We have modified BE-PUM in a way that instead of blindly follow any path, we only follow if there is a guarantee that the path is always taken for any initial environment. The judgment bases on a constant propagation analysis on the partial CFG, which is done on-the-fly during graph construction.

2. We have designed a constant propagation analysis that can be applied to X86 assembly programs (including memory and the stack, although they are restricted), and can be done on-the-fly together with BE-PUM execution.

3. The runs of BE-PUM with constant propagation application will eventually terminate, if the input programs are correctly implemented ,i.e. terminating for all input environments, and the set of the possible locations of the next instructions is finite, for any instruction in the program (this is due to the exhaustive destination check of BE-PUM).

4. Experiments have been done to compare the three versions: normal unrolling, convergence check and unrolling based on the result of constant propagation, on 1000 from real-world malware samples.

1.5 Outline for later sections

– In chapter 2, we will talk about the abstract interpretation and a brief idea of constant propagation analysis.
– Chapter 3 explains about x86 assembly programs, the environment and its operational semantics.
– BE-PUM architecture is explained in chapter 4, including the exhaustive check algorithm and related works.
– Chapter 5 is a detailed look at the constant propagation analysis, and current restriction when applying to X86 assemblies.
– Chapter 6 is the experimental data, and we summarize the thesis in chapter 7.

Chapter 2

Abstract Interpretation

2.1 Simple programs - Syntax and semantics

Regarding the programs with the following syntax:

2.1.1 Program syntax

Let $\mathbb{Z}$ be the set of numeric values, $\mathcal{V}$ the set of variables, and $\mathsf{Bool} = \{true, false\}$ the set of boolean values. We use metavariables n, x, and b for elements of $\mathbb{Z}, \mathcal{V}$ and Bool, respectively.

Definition 1. *The sets* Aexp *of arithmetic expressions,* Bexp *of boolean expressions and* Com *of program commands are defined as:*

$$\begin{aligned}
&\mathsf{Aexp} ::= n \mid x \mid \mathsf{Aexp} + \mathsf{Aexp} \mid \mathsf{Aexp} - \mathsf{Aexp} \mid \mathsf{Aexp} \times \mathsf{Aexp}^{1},\\
&\mathsf{Bexp} ::= b \mid \mathsf{Aexp} = \mathsf{Aexp} \mid \mathsf{Aexp} < \mathsf{Aexp} \mid \mathsf{Bexp} \wedge \mathsf{Bexp} \mid \mathsf{Bexp} \vee \mathsf{Bexp} \mid \neg\mathsf{Bexp}, \textit{ and}\\
&\mathsf{Com} ::= \mathsf{start} \mid \mathsf{exit} \mid \mathsf{skip} \mid x := \mathsf{Aexp}.
\end{aligned}$$

For an arithmetic expression $expr \in \mathsf{Aexp}$*, we write* $\mathsf{varRef}(expr)$ *for the set of all of its variables. A program is a 4-tuple* $\mathcal{P} = \langle \mathsf{Q}, \mathsf{q_0}, \mathsf{E}, \mathsf{L} \rangle$*, where*

- Q *is the set of locations,* $\mathsf{q_0} \in \mathsf{Q}$ *is the start location,*
- $\mathsf{E} \subseteq \mathsf{Q} \times \mathsf{Bexp} \times \mathsf{Q}$ *is the set of transition relations, and*
- $\mathsf{L} : \mathsf{Q} \to \mathsf{Com}$ *maps each location to its command, with* $\mathsf{L}(\mathsf{q_0}) = \mathsf{start}$.

Let $\mathsf{pred}, \mathsf{succ} : \mathsf{Q} \to 2^{\mathsf{Q}}$ *where:*

$$\begin{aligned}
\mathsf{pred}(q) &= \{q' \mid (q', q) \in \mathsf{E}\}, \textit{and}\\
\mathsf{succ}(q) &= \{q' \mid (q, q') \in \mathsf{E}\},
\end{aligned}$$

we require that for all $q \in \mathsf{Q}$*:*

- $\mathsf{pred}(\mathsf{q_0}) = \emptyset$*, and* $\mathsf{pred}(q) \neq \emptyset$ *if* $q \not\equiv \mathsf{q_0}$.
- *If* $\mathsf{L}(q) = \mathsf{exit}$ *then* $\mathsf{succ}(q) = \emptyset$*. Otherwise,* $\mathsf{succ}(q) \neq \emptyset$.

[1]We use abstract syntax and precedence does not take into account

2.1.2 Program semantics

Definition 2. *Let* $\mathsf{Env} = [\mathcal{V} \to \mathbb{Z}]$ *be the set of all functions from* $\mathcal{V}$ *to* $\mathbb{Z}$*. Given* $e \in \mathsf{Env}$*, the evaluations of arithmetic and boolean expressions are defined as* $\mathsf{aval} : \mathsf{Aexp} \times \mathsf{Env} \to \mathbb{Z}$ *and* $\mathsf{bval} : \mathsf{Bexp} \times \mathsf{Env} \to \mathsf{Bool}$*. Let* $i \in \mathbb{Z}$*,* $x \in \mathcal{V}$*,* $a_0, a_1 \in \mathsf{Aexp}$*,* $b, b_0, b_1 \in \mathsf{Bexp}$*, we have:*

$$\begin{aligned}
\mathsf{aval}(i, e) &= i,\ \mathsf{aval}(x, e) = e(x),\\
\mathsf{aval}(a_0 + a_1, e) &= \mathsf{aval}(a_0, e) + \mathsf{aval}(a_1, e),\\
\mathsf{aval}(a_0 - a_1, e) &= \mathsf{aval}(a_0, e) - \mathsf{aval}(a_1, e),\\
\mathsf{aval}(a_0 \times a_1, e) &= \mathsf{aval}(a_0, e) \times \mathsf{aval}(a_1, e),\\
\mathsf{bval}(\mathit{true}, e) &= \mathit{true},\ \mathsf{bval}(\mathit{false}, e) = false,\\
\mathsf{bval}(a_0 = a_1, e) &= \begin{cases} \mathit{true} & \mathit{if}\ \mathsf{aval}(a_0, e) = \mathsf{aval}(a_1, e),\\ \mathit{false} & \mathit{otherwise,} \end{cases}\\
\mathsf{bval}(a_0 < a_1), e &= \begin{cases} \mathit{true} & \mathit{if}\ \mathsf{aval}(a_0, e) < \mathsf{aval}(a_1, e),\\ \mathit{false} & \mathit{otherwise,} \end{cases}\\
\mathsf{bval}(b_0 \wedge b_1, e) &= \begin{cases} \mathit{true} & \mathit{if}\ \mathsf{bval}(b_0, e) = \mathsf{bval}(b_1, e) = \mathit{true},\\ \mathit{false} & \mathit{otherwise,} \end{cases}\\
\mathsf{bval}(b_0 \vee b_1, e) &= \begin{cases} \mathit{false} & \mathit{if}\ \mathsf{bval}(b_0, e) = \mathsf{bval}(b_1, e) = \mathit{false},\\ \mathit{true} & \mathit{otherwise,\ and} \end{cases}\\
\mathsf{bval}(\neg b, e) &= \begin{cases} \mathit{false} & \mathit{if}\ \mathsf{bval}(b, e) = \mathit{true},\\ \mathit{true} & \mathit{otherwise.} \end{cases}
\end{aligned}$$

Definition 3. *A state is an element of* $\mathsf{Q} \times \mathsf{Env}$ *describing the current location and the current environment. The initial state is* $\langle \mathsf{q_0}, \mathsf{e_0} \rangle$ *where* $\mathsf{e_0}$ *is the initial environment. The next state relation* $\to$ *describes how we move from one state to another in one step, and is defined as follows:*

$$\frac{(q, b, q') \in \mathsf{E} \quad \mathsf{bval}(b, e) = \mathit{true} \quad \mathsf{L}(q) \in \{\mathsf{start}, \mathsf{skip}\}}{\langle q, e \rangle \to \langle q', e \rangle}$$

$$\frac{(q, b, q') \in \mathsf{E} \quad \mathsf{bval}(b, e) = \mathit{true} \quad \mathsf{L}(q) = x := \mathit{expr}}{\langle q, e \rangle \to \langle q', e[\mathsf{aval}(\mathit{expr}, e)/x] \rangle}$$

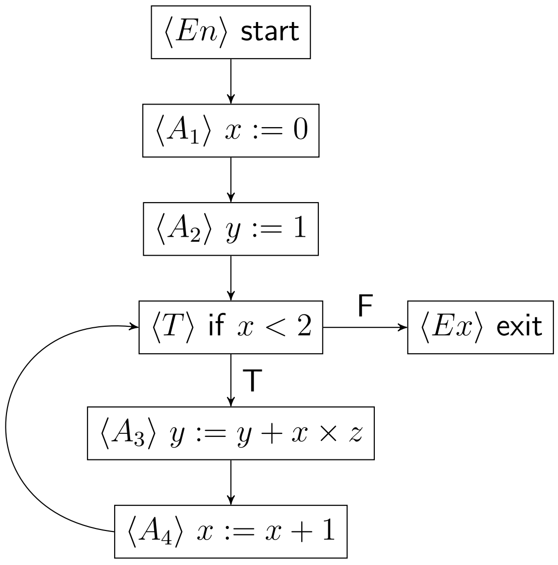
Given a program $\mathcal{P}$ *and an initial state* $s_0 = \langle \mathsf{q_0}, \mathsf{e_0} \rangle$*, the result of executing the program is an environment* e *such that we have* $\langle \mathsf{q_0}, \mathsf{e_0} \rangle \to^* \langle q, e \rangle$ *for some* $q \in \mathsf{Q}$ *&* $\mathsf{L}(q) = \mathsf{exit}$.

Example 1. Given a program $\mathcal{P}$ as in the below figure, where the initial state is

$$\langle En, \{(x, -1), (y, -1), (z, 2)\} \rangle$$

A transition (q, b, q') is represented as an edge with b is the edge label (which is omitted when $b = \mathit{true}$). The execution of the program starting from the initial state is:

$$\begin{array}{rll} & \langle En, & \{(x,-1),(y,-1),(z,2)\}\rangle \\ \to & \langle A_1, & \{(x,-1),(y,-1),(z,2)\}\rangle \\ \to & \langle A_2, & \{(x,0),(y,-1),(z,2)\}\rangle \\ \to & \langle T, & \{(x,0),(y,1),(z,2)\}\rangle \\ \to & \langle A_3, & \{(x,0),(y,1),(z,2)\}\rangle \\ \to & \langle A_4, & \{(x,0),(y,1),(z,2)\}\rangle \\ \to & \langle T, & \{(x,1),(y,1),(z,2)\}\rangle \\ \to & \langle A_3, & \{(x,1),(y,1),(z,2)\}\rangle \\ \to & \langle A_4, & \{(x,1),(y,3),(z,2)\}\rangle \\ \to & \langle T, & \{(x,2),(y,3),(z,2)\}\rangle \\ \to & \langle Ex, & \{(x,2),(y,3),(z,2)\}\rangle . \end{array}$$



Hence the result of executing the program is the environment $\{(x,2),(y,3),(z,2)\}$.

2.2 Abstract interpretation

Hereafter, if $\langle S, \sqsubseteq\rangle$ is an arbitrary lattice, we write $\sqcup$ for the least upper bound operator on S. Let $s_1, s_2 \in S$, we have

$$s_1 \sqcup s_2 = s \in S \text{ s.t. } s_1 \sqsubseteq s \ \&\ s_2 \sqsubseteq s \ \&\ (\forall s'.s_1 \sqsubseteq s' \ \&\ s_2 \sqsubseteq s' \implies s \sqsubseteq s')\,.$$

We denote $\overrightarrow{S} = [\mathsf{Q} \to S]$ with the ordering $sv \sqsubseteq sv'$ iff $\forall q \in \mathsf{Q}.\ sv(c) \sqsubseteq sv'(c)$. Clearly $\langle \overrightarrow{S}, \sqsubseteq\rangle$ is a lattice and it is complete if $\langle S, \sqsubseteq\rangle$ is complete.

Definition 4. *Given a program $\mathcal{P}$ with an initial state $s_0 = \langle \mathsf{q}_0, \mathsf{e}_0\rangle$, we define the context at some location q to be the set of all environments that may associate to q during computation sequences of $\mathcal{P}$. The context vector maps each location to a context. Let* $\mathsf{Cont} = 2^{\mathsf{Env}}$*, the context vector* CV *of $\mathcal{P}$ is defined as:*

$$\mathsf{CV} \in \overrightarrow{\mathsf{Cont}},\ \textit{where } \mathsf{CV}(q) = \{e \mid \exists n \geq 0.\langle \mathsf{q}_0, \mathsf{e}_0\rangle \to^* \langle q, e\rangle\}.$$

Example 2. The content of the context vector of the program in example 1 is:

$$\begin{array}{ll} (EN, & \{\{(x,-1),(y,-1),(z,2)\}\}) \\ (A_1, & \{\{(x,-1),(y,-1),(z,2)\}\}), (A_2, \{\{(x,0),(y,-1),(z,2)\}\}) \\ (T, & \{\{(x,0),(y,1),(z,2)\},\{(x,1),(y,1),(z,2)\},\{(x,2),(y,3),(z,2)\}\}) \\ (A_3, & \{\{(x,0),(y,1),(z,2)\},\{(x,1),(y,1),(z,2)\}\}) \\ (A_4, & \{\{(x,0),(y,1),(z,2)\},\{(x,1),(y,3),(z,2)\}\}) \\ (EX, & \{\{(x,2),(y,3),(z,2)\}\}). \end{array}$$

Definition 5. *An interpretation $\mathcal{I}$ of a program is a tuple $\mathcal{I} = \langle \textit{I-Cont}, \sqsubseteq, \textit{Int} \rangle$, where $\langle \textit{I-Cont}, \sqsubseteq \rangle$ is a complete lattice, and $\textit{Int} : \mathsf{Q} \times \overrightarrow{\textit{I-Cont}} \to \textit{I-Cont}$ is order-preserving. We define $\overrightarrow{\textit{Int}}$ to be a function between context vectors:*

$$\overrightarrow{\textit{Int}} : \overrightarrow{\textit{I-Cont}} \to \overrightarrow{\textit{I-Cont}}, \text{ with}$$
$$\overrightarrow{\textit{Int}}(cv) = \lambda q \in \mathsf{Q}.\textit{Int}(q, cv).$$

Clearly $\overrightarrow{\textit{Int}}$ is monotonic. By Tarski fixed point theorem, it has fixed points, and the set of all fixed points forms a complete lattice.

Definition 6. *Given two interpretations*

$$\mathcal{I} = \langle \textit{I-Cont}, \sqsubseteq, \textit{Int} \rangle, \text{ and } \mathcal{I}' = \langle \textit{I-Cont}, \sqsubseteq, \textit{Int} \rangle,$$

we say that $\mathcal{I}'$ is a consistent abstraction *of $\mathcal{I}$, iff there exist two functions α, γ called abstraction and concretization functions, such that:*

- $\alpha : \textit{I-Cont} \to \textit{I-Cont}'$, $\gamma : \textit{I-Cont}' \to \textit{I-Cont}$,
- *α and γ are order-preserving,*
- $\forall c' \in \textit{I-Cont}'.c' = \alpha(\gamma(c'))$,
- $\forall c \in \textit{I-Cont}.c \sqsubseteq \gamma(\alpha(c))$,
- *Let $\overrightarrow{\gamma} : \overrightarrow{\textit{Int}'} \to \overrightarrow{\textit{Int}}$ s.t. $\overrightarrow{\gamma}(cv') = \lambda q \in \mathsf{Q}.\gamma(cv'(q))$, then we have*

$$\textit{Int}(q, \overrightarrow{\gamma}(cv')) \sqsubseteq \gamma(\textit{Int}'(q, cv') \text{ for all } q \in \mathsf{Q}, cv' \in \overrightarrow{\textit{I-Cont}}'.$$

Lemma 1. *Let ℓ be least fixed point of $\overrightarrow{\textit{Int}}$, ℓ' be least fixed point of $\overrightarrow{\textit{Int}'}$, then*

$$\ell \sqsubseteq \overrightarrow{\gamma}(\ell').$$

Proof. Based on two observations:

$$\forall x \in \overrightarrow{\textit{I-Cont}}. \sqcap \{\ell, \overrightarrow{\textit{Int}}(x)\} \sqsubseteq x \Rightarrow \ell \sqsubseteq x, \text{ and}$$
$$\overrightarrow{\textit{Int}}(\overrightarrow{\gamma}(\ell')) \sqsubseteq \overrightarrow{\gamma}(\overrightarrow{\textit{Int}'}(\ell')) \sqsubseteq \overrightarrow{\gamma}(\ell'),$$

we have: $\sqcap\{\ell, \overrightarrow{\textit{Int}}(\overrightarrow{\gamma}(\ell'))\} \sqsubseteq \sqcap\{\ell, \overrightarrow{\gamma}(\ell')\} \sqsubseteq \overrightarrow{\gamma}(\ell') \Rightarrow \ell \sqsubseteq \overrightarrow{\gamma}(\ell')$. □

Our target is that by using abstract interpretation, it will be easier to find the least fixed point of $\overrightarrow{\textit{Int}'}$, which is then concretized into an approximation of the one in $\overrightarrow{\textit{Int}}$.

2.3 Constant Propagation

Abstract interpretation can be applied to static program analysis. In normal setup, the abstract domain is a finite cpo, hence the fixed point of the recursive function can be calculated in finite steps of iterations. Consider the abstract domain AVal which is a cpo with three elements: $\bot \sqsubseteq \mathsf{C} \sqsubseteq *$. The abstract environment $\mathsf{AEnv} = [\mathcal{V} \to \mathsf{AVal}]$ is attached to each location. Let e be the environment at a location q and $x \in \mathcal{V}$, then $e(x) = \mathsf{C}$ means that regardless of the initial environment, after executing on the same path, the value of x at q must be a constant: $\forall e_0, e'_0 \in \mathsf{Env}, x \in \mathcal{V}.e(x) = \mathsf{C}$ iff

$$\begin{aligned}\forall q_1, ..., q_n \in \mathsf{Q} \text{ s.t. } & \exists e_1, ..., e_n, e_c, e'_1, ..., e'_n, e'_q \text{ with}\\ & \langle \mathsf{q}_0, e_0\rangle \to \langle q_1, e_1\rangle \to \cdots \to \langle q_n, e_n\rangle \to \langle q, e_q\rangle \ \&\\ & \langle \mathsf{q}_0, e'_0\rangle \to \langle q_1, e'_1\rangle \to \cdots \to \langle q_n, e'_n\rangle \to \langle q, e'_q\rangle, \text{ then}\\ & e_q(x) = e'_q(x).\end{aligned}$$

Example 3. Consider the program in *Example* 1, let e be the constant propagation environment at T_1, then we have $e(x) = \mathsf{C}$, but $e(y) = e(z) = *$ because if we change initial value of z, the value of y and z at each iteration will also change.

Definition 7. *We define the constant propagation function* $\mathsf{cpInt} : \mathcal{V} \times \mathsf{Com} \times \mathsf{AEnv} \to \mathsf{AEnv}$*. Let* $x \in \mathcal{V}, c \in \mathsf{Com}, e \in \mathsf{AEnv}$*, we have:*

$$\begin{aligned}\mathsf{cpInt}(x, x := exp, e) &= \mathsf{C} \ \textit{if}\ \forall y \in \mathsf{varRef}(expr).e(y) = \mathsf{C},\\ \mathsf{cpInt}(x, x := exp, e) &= * \ \textit{if}\ \exists y \in \mathsf{varRef}(expr).e(y) = *,\\ \mathsf{cpInt}(x, c, e) &= e(x), \ \textit{otherwise.}\end{aligned}$$

Let $CP = \langle \mathsf{AEnv}, \sqsubseteq, \mathsf{AInt}\rangle$ *be the abstract interpretation of the program, where* $\mathsf{AInt} : \mathsf{Q} \times \overrightarrow{\mathsf{AEnv}} \to \mathsf{AEnv}$*. Let* $q \in \mathsf{Q}, cont \in \overrightarrow{\mathsf{AEnv}}$*, we have:*

$$\begin{aligned}\mathsf{AInt}(\mathsf{q}_0, cont) &= \lambda x \in \mathcal{V}.*, \ \textit{and}\\ \mathsf{AInt}(q, cont) &= \bigsqcup_{\substack{p \in \mathsf{pred}(q)\\ e = cont(p)}} \lambda x \in \mathcal{V}.\mathsf{cpInt}(x, L(p), e) \ \textit{if}\ q \not\equiv \mathsf{q}_0.\end{aligned}$$

From the definition of AInt, we have the lemma:

Lemma 2. *Let* CPCont *be the fixed point of the recursive function* $\mathcal{F} : \overrightarrow{\mathsf{AEnv}} \to \overrightarrow{\mathsf{AEnv}}$ *with*

$$\mathcal{F}(cont) = \lambda q \in \mathsf{Q}.\mathsf{AInt}(q, cont)$$

Then for any location $q \in \mathsf{Q}, x \in \mathcal{V}$*, we have* $\mathsf{CPCont}(q)(x) = \mathsf{C}$ *iff for any path that reaches* q*, the value of* x *is a constant.*

Chapter 3

X86 Assembly - Syntax and semantics

X86 assembly programs have some remarkable differences, comparing to other programs:

- Instructions (or commands) are a part of the environment, and will be loaded during computation. Such instructions can also change the environment, which will affect the next instruction to be loaded. The initial environment will contain the information of the first instruction to be executed.
- We have the mechanism to handle unwanted behaviors, i.e. parsing error, un-allowed operations, ... through exception handling, which will change the normal control flow and may be used to cheat the analyzer.
- There is a special segment in the memory that is implemented as a stack (which supports push and pop operations). Calls will store the address of the return point in the stack. One remarkable thing is that, in general, we can freely modify the stack content as normal memory locations.

3.1 Syntactic sets

Definition 8. *Let* $\mathsf{Loc} \subseteq \mathbb{N}$ *be the set of instruction locations,* $\mathsf{Mem} \subseteq \mathbb{N}$ *the set of memory locations, with* $m \in \mathsf{Mem}$ *indicates the memory* of one byte starting at address *m, and* Regs, Flags *the sets of general purpose and flag registers. We denote* $\mathcal{V} = \mathsf{Regs} \cup \mathsf{Flags} \cup \mathsf{Mem}$ *to be the set of program variables, and* $\mathsf{Expr} = \mathsf{Aexp} \cup \mathsf{Bexp}$ *the set of arithmetic and boolean expressions defined based on* $\mathcal{V}$*.*

During execution, values are described as a bit (for boolean value) or a sequence of bits. Each instruction defines the number of bits it uses, which need to have the same type with all of its arguments. For example, movw ax, [0x4010030] will move the value of 2-byte register ax to memory locations [0x403000] and [0x403001], while movl eax, [0x403000] modifies 4 bytes of memory. Given that the addresses are valid in data segment, consider the example:

```
movl eax, 0             ; eax is initialized into (0x) 00 00 00 00
movl [0x403000], eax    ; memory image starting from [0x4010030]: (0x) 00 00 00 00
movb al, −128           ; al: (0x) 80, eax now becomes (0x) 00 00 00 80
movb [0x403002], al     ; memdump from [0x4010030]: (0x) 00 00 80 00
movl eax, [0x403000]    ; eax now has the value (0x) 00 80 00 00 = +8388608
                        ; (due to LSB storage mechanism of x86 machines).
```

Definition 9. *Let $n \in \mathbb{N}$, the stack is defined as $\mathsf{Stack} = \bot \mid n : \mathsf{Stack}$ with each stack entry indicates a stack element of 16-bit size (smallest size of elements in stack). Let $S_1, S_2 \in \mathsf{Stack}$, we have $S_1 \sqsubseteq S_2$ iff*

$$(S_1 \equiv \bot) \text{ or } \big(S_1 \equiv n : S_1' \ \&\ S_2 \equiv m : S_2' \ \&\ n \sqsubseteq m \ \&\ S_1' \sqsubseteq S_2'\big).$$

Let $\mathsf{Inst} = \mathsf{AInst} \cup \mathsf{CInst}$ be the set of instructions, with each instruction $I \in \mathsf{Inst}$ containing the information of:

- *Instruction name, its size in memory (2 bytes, 3 bytes, ...),*
- *Executing condition (mostly for control instructions),*
- *Size of arguments (8-bit, 16-bit, or 32-bit), and*
- *Number of arguments, argument information.*

There are two kinds of instructions: data (AInst) and control instructions (CInst). The main different is how the system identify the next instruction afterward. After a data instruction, we go to the successive instruction in the memory, while after a control instruction, the location of next instruction may depend on current context. Control instructions consist of jumps and calls (jump to a known-address of an API is considered a call). There are two kinds of calls: system interrupts and API calls:

- System interrupts occur while system detects an invalid operation (decoding with invalid opcode, priviledge violations, pagefault, hardware interrupts, ...), or a programmed exception (via INT instruction and its relative, mostly involving DOS API). This is outdated, and in modern systems, we tend to use system API instead. However, some of them may still be used.
- API calls can be done by jumping directly to a known address of an API or via function symbols that are declared in the import table. System will calculate the target address of the callee, push the return address (address of successive instruction) to stack and jump to the target address. The callee may or may not return to the caller (depending on which API is in charge), but in such case, the return address will be popped out and control is passed to the instruction at this address, together with current environment.

Our target programs are those which are correctly implemented, and unexpected exceptions do not occur. Furthermore, system API calls will be executed correctly as described in reference (whether it returns, and how the environment is modified during execution), and although in general, stack is a modifiable segment in the memory, we assume that it can only be accessed via stack related instructions, i.e. push, pop, enter, leave and their relatives.

3.2 Semantics

Definition 10. *Let* $\mathsf{Env} = [\mathsf{Regs} \to \mathbb{Z}] \times [\mathsf{Flags} \to \mathsf{Bool}] \times \mathsf{Stack} \times [\mathsf{Mem} \to \mathbb{Z}]$ *be the set of environments. Assume we are given the decoding and auxiliary functions:*

- $\mathsf{instAt} : \mathsf{Env} \times \mathsf{Loc} \to \mathsf{Inst}$ *(decoding function),*
- $\mathsf{sizeOf} : \mathsf{Inst} \to \mathbb{N}$ *returns the instruction size,*
- $\mathsf{cond} : \mathsf{CInst} \to \mathsf{Bexp}$ *returns the condition, and*
- $\mathsf{dest} : \mathsf{CInst} \to \mathsf{Aexp}$ *returns the destination of a control instruction.*

The semantic function $[\![.]\!]$ *defines instruction execution and expression evaluation:*

$$[\![.]\!] \subseteq (\mathsf{Inst} \times \mathsf{Env} \to \mathsf{Env}) \cup (\mathsf{Aexp} \times \mathsf{Env} \to \mathbb{Z}) \cup (\mathsf{Bexp} \times \mathsf{Env} \to \mathsf{Bool}).$$

Let $\mathsf{State} = \mathsf{Loc} \times \mathsf{Inst} \times \mathsf{Env}$ *be the set of states. Given initial state* (ℓ_0, I_0, σ_0)*, the state transition relation* $\to$ *is defined as follows:*

$$\frac{I \in \mathsf{AInst} \quad \sigma' = [\![I, \sigma]\!] \quad \ell' = \ell + \mathsf{sizeof}(I) \quad I' = \mathsf{instAt}(\sigma', \ell')}{(\ell, I, \sigma) \to (\ell', I', \sigma')}$$

$$\frac{I \in \mathsf{CInst} \quad [\![\mathsf{cond}(I), \sigma]\!] = \mathit{true} \quad \sigma' = [\![I, \sigma]\!] \quad \ell' = [\![\mathsf{dest}(I), \sigma']\!] \quad I' = \mathsf{instAt}(\sigma', \ell')}{(\ell, I, \sigma) \to (\ell', I', \sigma')}$$

$$\frac{I \in \mathsf{CInst} \quad [\![\mathsf{cond}(I), \sigma]\!] = \mathit{false} \quad \sigma' = [\![I, \sigma]\!] \quad \ell' = \ell + \mathsf{sizeof}(I) \quad I' = \mathsf{instAt}(\sigma', \ell')}{(\ell, I, \sigma) \to (\ell', I', \sigma')}.$$

The CFG of a program is the graph representation of all paths that may occur during program execution (i.e. state transitions). We have $\mathsf{Graph} = \mathsf{Node} \times \mathsf{E}$*, with* $\mathsf{Node} = \mathsf{Loc} \times \mathsf{Inst}$ *and* $\mathsf{E} \subseteq \mathsf{Node} \times \mathsf{Node}$*. Starting node is* (ℓ_0, I_0)*. We have the property that for any execution sequence*

$$(\ell_0, I_0, \sigma_0) \to (\ell_1, I_1, \sigma_1) \to \cdots \to (\ell_n, I_n, \sigma_n), n \in \mathbb{N}$$

there exists a corresponding path in the graph: $(\ell_0, I_0) \to (\ell_1, I_1) \to \cdots \to (\ell_n, I_n)$*.*

Given the binary program, BE-PUM tries to generate the CFG on-the-fly using concolic testing. It tries to generate test cases to cover all possible paths and uses a simulator to check for viability. The method involves loop unrolling and exhausting control destination searching. However, one problem is that in general, the initial environment is not fixed, and in some specific cases, BE-PUM will be stuck in an infinite loop of finding an instance to cover the new paths. We hope that using constant propagation, we can reduce such problems.

Chapter 4

BE-PUM

4.1 CFG generation

Due to the nature of X86 assemblies, the CFG of an assembly program can only be built on-the-fly, and we have to execute the program in some way. *Symbolic execution* is a traditional technique to symbolically execute a program, which maintains a symbolic state (ℓ, ρ) where ℓ is a program location and ρ is the path formula of the path from program entry point to ℓ. A path formula ρ describes the precondition of the execution path to ℓ, starting from the precondition at the program entry. If ρ is satisfiable (often checked by SAT/SMT solvers), the path is *feasible*. There are two typical methodologies:

– **Static symbolic execution (SSE)**, in which the feasibility of an execution path is checked by satisfiability of ρ.

– **Dynamic symbolic execution (DSE)**, in which the feasibility is checked by testing with a satisfiable instance of ρ (*concolic testing*). For detailed testing (e.g. stepwise execution, trace analysis), it requires an emulator.

A path formula often describes a restricted view of a program. For instance, system status, kernel procedures and memory offsets are often omitted. Furthermore, sometimes formulas are restricted to decidable logic (e.g. Presburger arithmetic), a path formula would be an approximation. In such cases, DSE gives more precise results.

In BE-PUM, CFG generation proceeds with symbolic execution, and is a saturation procedure on configurations of the form $\langle P, \Delta, \psi \rangle$ described in definition 11.

Definition 11. *Let*

$$\mathsf{Conf} = 2^{\mathsf{Node}} \times 2^{\mathsf{E}} \times (\mathsf{Node} \times \mathsf{SymEnv} \times \mathsf{Pred})$$

be the set of configurations, with SymEnv *is the set of symbolic environments and* Pred *is the set of path formulas. During symbolic execution, a configuration $\langle N, E, \langle (\ell, I), e, \rho \rangle \rangle$ holds the value of the current graph and the symbolic state.*

Starting from initial configuration $\langle \{(\ell_0, I_0)\}, \emptyset, \langle (\ell_0, I_0), e_0, true \rangle \rangle$, the CFG generation process proceeds with the rule $\langle N, E, \langle (\ell, I), e, \rho \rangle \rangle \vdash \langle N', E', \langle (\ell', I'), e, \rho' \rangle \rangle$, if there is a transition $(\ell, I, e) \rightarrow (\ell', I', e')$ and

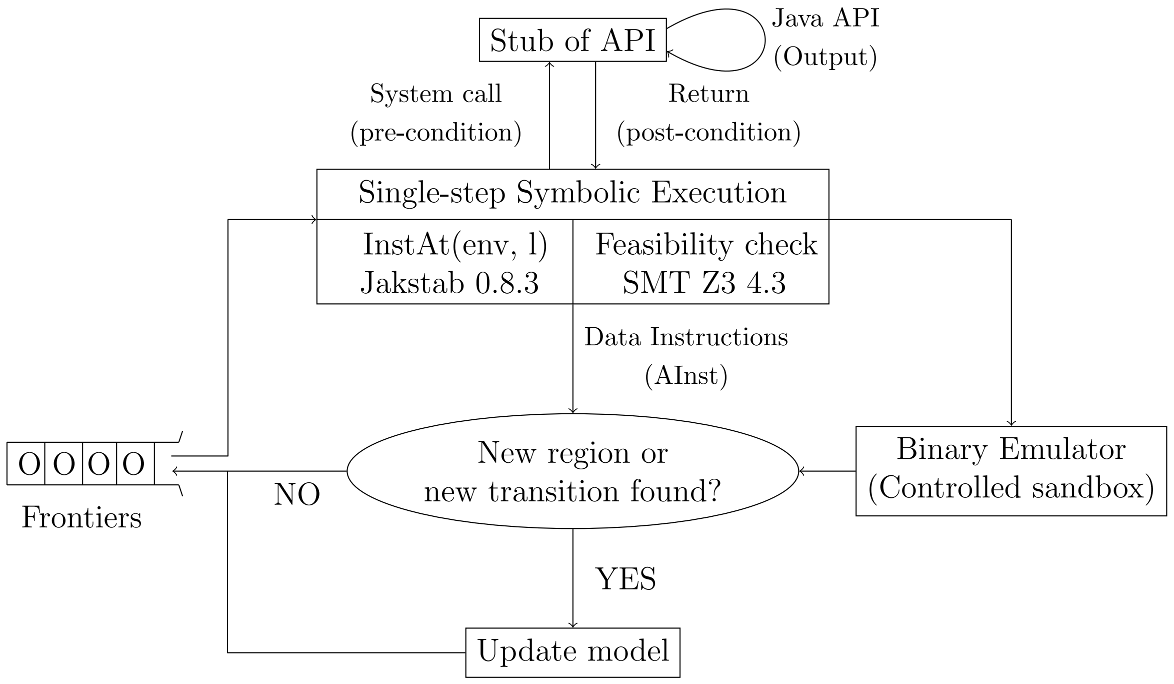


Figure 4.1: The BE-PUM architecture

- $N' = N \cup \{(\ell', I')\}$,
- $E' = E \cup \{(\ell, I) \rightarrow (\ell', I')\}$,
- $\rho' = \mathsf{post}(\rho) \wedge \mathsf{SideCond}$, *with* $\mathsf{post}(\rho)$ *is the post condition at* (ℓ, I) *w.r.t precondition* ρ, *and* $\mathsf{SideCond}$ *is the side condition appears in the transition.*

The CFG generation will continue until the graph being stable. Since this process should be finitely bounded, complete CFG generation requires methods such as loop invariant generation. Currently BE-PUM simply unrolls the loops and puts the bound on the maximum number of times of unrolling; hence the result may be an under-approximation.

4.2 BE-PUM architecture

BE-PUM implements on-the-fly model generation based on symbolic execution. It applies *Jakstab 0.8.3* as a single-step disassembler, and *SMT Z3 4.3* as the backend engine to check the feasibility of execution paths. Figure 4.1 shows the architecture of BE-PUM, which consists of three components: *(single-step) symbolic execution*, *binary emulation*, and *pushdown model*. The symbolic execution picks up one from the frontiers (symbolic states at the ends of explored execution paths) in depth-first manner, and it tries to extend one step. If the instruction is a data instruction (i.e., only update the environment and the next location is statically decided), it will simply disassemble the next instruction. If the instruction is a control instruction (e.g., indirect jumps), concolic testing (at the binary emulator) is applied to decide the next location. In any case, either a new symbolic state or a new transition is found, they are stored in the model and a symbolic state is added to the frontiers. This procedure continues until either the exploration has converged, or reaching to unknown instructions, system calls, or out-of-bound addresses.

BE-PUM also implements stubs for (part of) Win32 APIs. Each system call is treated as a single instruction such that (i) the update of path formulas and environments follow to the technical specification at Microsoft Developer Network.[1]; 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. BE-PUM has successfully covered the most frequent 80 APIs in the VX Heaven database.

Remark 3. 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 item bases 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*

Remark 4. Initially, the system flags are randomly set for flag-dependent x86 instructions like `cmp` and `cjump`. This holds an under-approximation, since Win32 will generate any Boolean combination of system flags.

4.3 Exhaustive checking for indirect destination

The destination of a control instruction may depend on the initial environment, and there are many locations corresponding to many difference situations, BE-PUM tries to cover as much as possible.
Typical example is the assembly version of switch-case. In the sample program, we assume that instruction sizes are fixed to be 1 byte. The first four instructions do a check on initial value of `eax` and call the API `ExitProcess` to terminate the program if it is out of bound. Then depending on the value to be 0, 1 or 2, the jump at line 7 will jump to the instruction at line 8, 9, or 10, which may have difference behaviors.

Sample switch-case:

```
 1 cmpl eax, 0
 2 jl EXIT
 3 cmpl eax, 2
 4 jg EXIT
 5 movl ebx, OFFSET ADDR
 6 addl ebx, eax
 7 jmp ebx
   ADDR:
 8 jmp Case_0
 9 jmp Case_1
10 jmp Case_2
   EXIT:
11 call ExitProcess
```

Given ρ the path formula, the SMT solver will result a SAT instance (assignments on symbolic variables, which is the values of variables of the initial environment), and hence the initial environment, that satisfies ρ. Assume we are given the solving function

[1] `http://msdn.microsoft.com/library/`

$\mathsf{solve} : \mathsf{Pred} \to (\mathsf{Env} \cup \{\perp\})$ that returns $\perp$ if ρ is UNSAT. We will follow algorithm 1 to find all possible destinations of a control instruction.

Algorithm 1: Exhaustive destination search

```
   Data: $\langle I, \rho \rangle$: current symbolic state, $I \in \mathsf{CInst}$
   Result: Set $S$ of SAT instances that satisfy $\rho$
 1 $S := \emptyset$;
 2 while true do
 3     $e := \mathsf{solve}(\rho)$;
 4     if $e = \perp$ then
 5         break;
 6     else
 7         $S := S \cup \{e\}$;
 8         $\ell = [\![\mathsf{dest}(I), e]\!]$;
 9         $\rho := \rho \wedge (\mathsf{dest}(c) \neq \ell)$;
10     end
11 end
12 return S;
```

Remark 5. In the algorithm, the location of the next instruction is identified based on the symbolic environment. Each SAT instance corresponds to one case of jump destinations. However the destination resulted after dynamically running with such SAT instance may be different (an exception occurred and control transferred to the exception handler, for instance). Currently we assume that such problems do not occur.

4.4 Related works

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 code for obfuscation. For the former, a main obstruction is scalability. *IDA Pro* [2] is the most popular disassembler, combining recursive disassembly and linear sweep.

There are various model generation tools for binary executable. Detection of destinations of indirect jumps can be done in either static or dynamic method. BE-PUM stands between, using both methods. For instance, CodeSurfer/x86 [1], McVeto [9], and JakStab [8] [7] adopt a static approach, while Syman [5] chooses a dynamic one. Static methods are abstract interpretation (static analysis) for an over approximation, and static symbolic execution to check feasibility of an execution path for an under-approximation. JakStab and McVeto apply CEGAR [3] to refine the over approximations. McVeto also uses static symbolic execution to check the path feasibility (named a *concrete trace*).

[2] `https://www.hex-rays.com/products/ida/`

Dynamic methods are various testing, e.g., random testing, dynamic symbolic execution (concolic testing). Note that

- For deciding destinations, it requires stepwise execution. Thus, dynamic methods require binary emulators.
- For symbolic execution, states of a model require certain abstraction on memory images. Otherwise path conditions relate every system detail.

Syman tries to emulate the full Windows OS. Such detailed information makes symbolic execution complex and models easily explode even for small binary code. BE-PUM and McVeto extract only control structures (and detailed information will be recovered by model checking), and apply pushdown models, which further reduce complexity of models.

For handling self-modification, on-the-fly model generation and stepwise disassembly are required [2]. JakStab, McVeto, Syman, and BE-PUM do them.

As 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). In the meantime, 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, we do not compare with experiments. However, targeting malware, BE-PUM handles indirect jump, instruction overlapping and self-modifying, as well as some stub APIs often used by malware, especially for SEH techniques. This is not surprising since dynamic methods are observed to be more effective on malware analysis [4].

It is also notably remarkable that even though there are various works aiming at producing model from binary code, BE-PUM is solely focusing on dealing with malware obfuscation. Thus, BE-PUM cares more on handling indirect jumps and self-modification and less on scalability, since most of malware are quite small (often less than 1K loc). Finally, BE-PUM targets on malware among binary codes (Syman as well), not system software. Thus, BE-PUM cares more on handling indirect jumps and self-modification and less on scalability, since most of malware are quite small (often less than 1K loc).

Chapter 5

Constant Propagation on BE-PUM

5.1 Problems

Consider the following program:

```
0x401000:   cmpl ecx, 0
0x401003:   jle 0x401008
0x401005:   decl ecx
0x401006:   jmp 0x4010050
0x401008:   call ExitProcess
```

The program starts from the instruction at `0x401000`, and `ExitProcess` is a system API that terminates the program and passes the control back to the operating system. Here we implicitly use `ecx` as an input parameter. Before it executes, the system will move the input value to `ecx` and jump to the start instruction. Depending on such value, call n, we will go through the loop n times (for simplicity, we only decrease `ecx` and do nothing else within the loop). We have a conditional jump at location `0x401003`, where the location of the next instruction may be `0x401005` or `0x401008`, depending on the current value of `ecx`. For each path, BE-PUM will check for satisfiability of the path condition. If it is SAT, the path is viable and BE-PUM will continue the analyzing process [1]. The above program has one problem that the number of times of loop unrolling is not explicitly bounded in the code. In general, the value of `ecx` ranges between -2^{31} and $2^{31} - 1$. BE-PUM will blindly follow the loop without finding any new instruction.

By applying constant propagation analysis on the partial CFG, we know that the condition of the instruction at `0x401003` is not a constant. Our approach is that if a jump revisits a node that was explored in the graph, we will continue checking the node only if the condition is a constant. This process is similar to loop unrolling, but we only unroll if there is a guarantee that the normal execution will also follow the same path, and since we have assumed that input programs are terminating, the unrolling process will also terminate.

[1]BE-PUM v2 uses embedded emulator, and next address is calculated from the SAT instance

5.2 Constant propagation for X86 assemblies

We will extend the analysis in section 2.3 to apply to BE-PUM.

Definition 12. *The set of abstract values* AVal *consists of three elements with the order:* $\bot \sqsubseteq \mathsf{C} \sqsubseteq *$. *Abstract environment* AEnv *is the cpo:*

$$\mathsf{AEnv} = [\mathsf{Regs} \to \mathsf{AVal}] \times [\mathsf{Flags} \to \mathsf{AVal}] \times \mathsf{AStack} \times [\mathsf{Mem} \to \mathsf{AVal}], \ \textit{where}$$

$$\mathsf{AStack} = \mathsf{AVal} : \mathsf{AStack} \mid \bot.$$

Let $S_1, S_2 \in \mathsf{AStack}$*, we have the relation*

$$\begin{aligned} S_1 \sqsubseteq S_2 \ \textit{iff}\ S_1 = \bot \ \textit{or} \\ (S_1 = n \mid S_1') \ \&\ (S_2 = m \mid S_2') \ \&\ n \sqsubseteq m \ \&\ S_1' \sqsubseteq S_2'. \end{aligned}$$

For each instruction, we define the functions need and mod that returns the set of variables or stack operations that are referenced or modified during executions:

$$\mathsf{need}, \mathsf{mod} : \mathsf{Inst} \to 2^{\mathcal{V}\,\cup\,\mathsf{StackOps}}, \text{ with } \mathsf{StackOps} = \{\mathsf{push}, \mathsf{pop}\},$$

and $\mathsf{envMod} : \mathsf{Inst} \times \mathsf{AEnv} \times 2^{\mathcal{V}\,\cup\,\mathsf{StackOps}} \times \mathsf{AVal} \to \mathsf{AEnv}$ results the modified environment w.r.t each operation:

$$\begin{aligned} \mathsf{envMod}(I, e, \emptyset, v) &= e, \\ \mathsf{envMod}(I, e, \mathsf{push} \cup S, v) &= \mathsf{envMod}(I, e[(v \mid e.stack)/e.stack], S, v), \\ \mathsf{envMod}(I, e, \mathsf{pop} \cup S, v) &= \mathsf{envMod}(I, e, S, v) \text{ if } e.stack = \bot\ , \\ \mathsf{envMod}(I, e, \mathsf{pop} \cup S, v) &= \mathsf{envMod}(I, e[S'/e.stack], S, v) \text{ if } e.stack = v' \mid S'\ , \\ \mathsf{envMod}(I, e, \{x\} \cup S, v) &= \mathsf{envMod}(I, e[v/x], S, v). \end{aligned}$$

Abstract interpretation for each instruction $\mathsf{aInt} : \mathsf{Inst} \times \mathsf{AEnv} \to \mathsf{AEnv}$ is defined as:

$$\mathsf{aInt}(I, e) = \mathsf{envMod}(I, e, \mathsf{mod}(I), v), \text{ where } v = \sqcup\{e(x) \mid x \in \mathsf{need}(I)\}.$$

We can prove that aInt is continuous.

We define the environment vector that maps each node in the partial graph with its abstract environment. Let $\overrightarrow{e} : \mathsf{Node} \to \mathsf{AEnv}$, $n, n' \in \mathsf{Node}$, we define the recursive function on the partial graph:

Definition 13. *Let* $\mathcal{F} : [\mathsf{Node} \to \mathsf{AEnv}] \to [\mathsf{Node} \to \mathsf{AEnv}]$*, with*

$$\mathcal{F}(\overrightarrow{e})(n) = \bigsqcup_{(n',n)\,\in\,\mathsf{E}} \mathsf{aInt}(n', \overrightarrow{e}(n')).$$

Then $\mathcal{F}$ *is continuous, and* $\mathit{fix}(\mathcal{F}) = \bigsqcup_{n \in \mathbb{N}} \mathcal{F}^n(\overrightarrow{\mathsf{e}_0})$ *where* $\overrightarrow{\mathsf{e}_0}$ *is the initial environment mapping.*

We will calculate $\mathit{fix}(\mathcal{F})$ w.r.t the partial graph when a new edge is added. During execution, if a conditional jump goes to the node that was already explored, instead of blindly unrolling, we only do that when the condition of the indirect jump is a contant w.r.t $\mathit{fix}(\mathcal{F})$. Since we have assumed that input programs are terminating and there are no infinite execution paths, the CFG generation will also terminate.

5.3 Model restriction

Because of the complexity of X86 assemblies and the strength of analysis, currently we have some restrictions:

5.3.1 Memory operands

In X86 instructions, memory location can be accessed indirectly via offsets. Consider the piece of code that move the value of ecx to 4 bytes memory starting from 0x0454DC:

```
movl eax, 0x4054DC
movl (eax), ecx
```

The second instruction cannot be correctly supported only with the information obtained by the abstract environment. An alternative way is to store the concrete value during BE-PUM execution, but there are still problems with loops and symbolic values. Since we don't have any concrete idea on the solution, currently such kind of operands is not supported.

5.3.2 The stack

The stack is handled by two registers: esp, the current stack pointer, and ebp, the base pointer of the current stack frame. There is a calling convention that when a function is called, typically space is reserved on the stack for local variables. This space is usually referenced via ebp (all local variables and function parameters are a known constant offset from this register for the duration of the function call.) esp, on the other hand, will change during the function call as other functions are called, or as temporary stack space is used for partial operation results. For instance, before a function call there may have some instructions like this:

```
push ebp
movl ebp, esp
movl esp, 20
```

The code will preserve 20-byte space for local variables on the stack. Such variables later can be referenced via offsets to ebp: -4(ebp), -16(ebp), … In general, the stack is a part of the memory, and we can freely modify the value of ebp and esp registers, hence modify the stack area. For instance, instead of a "push eax" instruction, we can do something like

```
subl esp, 4
movl (esp), eax
```

or write directly into the memory location of esp. Due to system complexity, we will regard the stack as an independent area, and can only be modified directly via stack operations, i.e. push, pop, call and ret. More specifically, indirect access to stack is not supported.

5.3.3 Self loop instructions

Example 4. Consider the example that copies *myStr* into *myStr2*:

```
section   .code
          movl esi, myStr
          movl edi, myStr2
          cld
          movl ecx, 4
          rep movsw                  ; mystr2 is now AaBbCca\0
section   .bss                       ; uninitialized data
          myStr2: resb 8
section   .data
          myStr db "AaBbCca", 0x0
```

The instruction rep movsw will repeat the move operation until ecx is zero. Since we don't know the concrete value of ecx, the impact to the memory cannot be correctly handled, currently we will skip such instructions.

The above restrictions come from the limitation of the analysis: the defined abstract domain is too coarse to handle all operations. Since our aim is that constant propagation is implemented as a module to BE-PUM and will not affect the performance heavily, meanwhile we will follow such configuration.

5.4 Interaction with BE-PUM

As stated in section 5.2, we will calculate the constant propagation environment when there is a new edge added to the graph. The initial graph contains the start node with the initial environment $\mathsf{e_0}$.

Remark 6. In the implementation, $\mathsf{e_0}$ will be defined to follow BE-PUM convention:

$$\begin{array}{ll} \mathsf{e_0}(x) = \mathsf{C} & \text{if } x \in \{\mathsf{edx}, \mathsf{esi}, \mathsf{edi}\}, \\ \mathsf{e_0}(N) = \mathsf{C} & \text{if } N \text{ is an address within data section of the binary program,} \\ \mathsf{e_0}(x) = * & \text{otherwise.} \end{array}$$

When a new edge is added to the graph $\langle \mathsf{Node}, \mathsf{E} \rangle$, we will re-calculate the abstract environment w.r.t the new one $\langle \mathsf{Node}', \mathsf{E}' \rangle$. Let $\mathcal{F}$ and $\mathcal{F}'$ be the recursive functions w.r.t the old and the new graph, respectively, we have the lemma:

Lemma 3. *Let $\overrightarrow{e}$ and $\overrightarrow{e'}$ be the fixpoints of $\mathcal{F}$ and $\mathcal{F}'$, respectively:*

$$\overrightarrow{e} = \mathit{fix}(\mathcal{F}) = \bigsqcup_{n \in \mathbb{N}} \mathcal{F}^n(\overrightarrow{\mathsf{e_0}}), \text{ and}$$
$$\overrightarrow{e'} = \mathit{fix}(\mathcal{F}') = \bigsqcup_{n \in \mathbb{N}} \mathcal{F}'^n(\overrightarrow{\mathsf{e_0}}),$$

then $\overrightarrow{e'}$ can be calculated via $\overrightarrow{e}$:

$$\overrightarrow{e'} = \bigsqcup_{n\in\mathbb{N}} \mathcal{F}'^n(\overrightarrow{e}).$$

Proof. Let $X = \bigsqcup_{n\in\mathbb{N}} \mathcal{F}'^n(\overrightarrow{e})$. We have $\overrightarrow{e'} \sqsubseteq X$ since $\mathcal{F}$ and $\mathcal{F}'$ are monotonic. The reverse direction can be proved starting from the observation that $\overrightarrow{e} \sqsubseteq \overrightarrow{e'}$ (the partial graph always becomes bigger during construction: $\mathsf{Node} \subseteq \mathsf{Node}'$ and $\mathsf{E} \subseteq \mathsf{E}'$). □

5.5 Termination

The fixed point operation defined in definition 13 does not terminate even we have assumed that the input programs eventually terminate. Consider the example that within the loop, it pushes an element into the stack without popping out:

```
start:  movl ecx, 3         ; for simplicity we initialize the loop counter
 loop:  cmpl ecx, 0         ; with a fixed constant
        je EXIT
        decl ecx
        push 0              ; put a constant value to the stack
        jmp loop
 EXIT:  call ExitProcess
```

The recursive function never terminates as it always increases the stack after each iteration. To ensure termination, in definition 12 we have to modify the stack configuration:

Definition 14. *(Refined configuration)*
The abstract stack AStack *is defined to be*

$$\mathsf{AStack} = \mathsf{AVal} : \mathsf{AStack} \mid \omega.$$

Let $S_1, S_2 \in$ AStack, we have the relation

$$S_1 \sqsubseteq S_2 \text{ iff } S_2 = \omega \text{ or } (S_1 = n \mid S_1') \;\&\; (S_2 = m \mid S_2') \;\&\; n \sqsubseteq m \;\&\; S_1' \sqsubseteq S_2'.$$

The function envMod *will be modified in the way that popping from the ω stack will results a $*$ value. Let $e, e' \in$ AEnv, if they differ only from the stack element: $e.stack \sqsubset e'.stack$ or vice versa, then*

$$e \sqcup e' = e[\omega/e.stack].$$

Remark 7. Note that the refinement will result an over-approximation and need further refinement. In the implementation, we only apply the join operator in definition 14 to reset the stack when other elements in the environment became stable.

Chapter 6

Experimental result

The experiments are performed on *Windows 7 machine* with *Intel Core i3-2359M CPU @2.30 GHz* and *4GB memory*, running *Java 1.8.0_31*. We compare between three versions:

1. BE-PUM with loop unrolling (i.e. we continue exploring the path even when it goes to an already explored location)
2. BE-PUM where we only unroll if the condition of a conditional jump is a constant. The constant propagation environment will be calculated when a new edge is added to the partial graph.
3. BE-PUM where loop unrolling is replaced by convergence check: the path is skipped if it goes to an already explored location.

on 1000 real-world malwares of 733 families[1] from the database of VX Heaven. Manual testing on small samples will be done using *OllyDBG v1.10*.

6.1 Assumptions

Besides the restrictions described in section 5.3, we have some more assumptions:

1. Since there are no available verifiers for the correctness of the generated models, in the mean time we have to assume that the actual runs will follow what were described in the documents[2], and the information resulted from the runs of OllyDBG is correct.
2. We do not cover system exception handlers. For such assumption, after a data instruction $I \in \mathsf{AInst}$, the control is always passed to the successive instruction and except being described in $\mathsf{mod}(I)$, variables will never be changed.
3. The inputs of constant propagation are the on-the-fly CFGs of BE-PUM, our claims will be applied for such partial CFGs.

[1]Each family originates from a single source base and exhibits a set of consistent behaviors
[2]Intel(R) 64 and IA-32 Architectures Software Developer Manuals

6.2 Halting condition

BE-PUM will stop the path with an *unknown instruction or API.* As it tries to follow all possible paths, to prevent infinite loop unrolling we will stop the path when a location is visited more than 1000 times, and the time-out is set to be 100s. There is a remark that we used the same configuration for all the runs.

Remark 8. Besides the restrictions mentioned in previous sections, we have some remarks on the implementation:

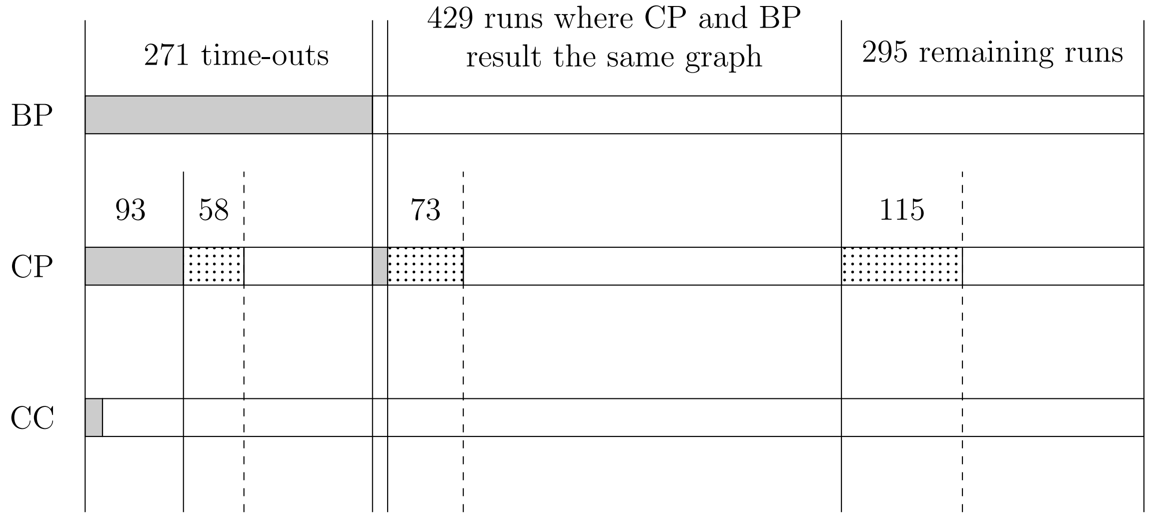
- Currently our implementation do not support API calls.
- We only unroll the loop when the jump condition (if available) is a constant. For other cases ($\perp$ or *), the path will be skipped.

6.3 Experimental results

As described in section 6.2, we will compare the three versions:

- `BP`: BE-PUM with normal loop unrolling.
- `CP`: BE-PUM which only unrolls if the condition of a conditional jump is a constant
- `CC`: Convergence check, a path is skipped if it revisits a node.

Doing a run on 1000 chosen samples on both versions of BE-PUM, we have:



1. BE-PUM has 271 time-out samples:
   - On the same set, `CP` has 93 time-outs and `CC` has 16 time-outs.
   - On 178 samples that both `CP` and `CC` finish, there are 58 cases that `CP` has better result (the CFG) than `CC`.

2. There are 4 cases that CP result time-out while both BP and CC finish (and they finish with the same graph).

3. We have 429 runs where BP and CP result the same graph. The number for BP and CC is 356.

4. On remaining samples, there are 115 cases that CP has better results than CC.

Since CP is a *better* choice than CC, there is no surprise that CP always results the better graph if it terminates (907/1000 samples). Comparing CP and BP, there are 299 cases that BP finishes with better results than CP. This is expected since CP results an under-approximation of the CFG. We only unroll the loop if there is a guarantee that the path is taken for any initial environment. There have the cases that the skipped paths eventually go to new locations. Some malwares start with the initialization of the variables:

```
xor eax, eax
```

follows by a conditional jump. Our configuration on the abstract domain fails for such kind of problems. For the concrete answers, refinements such as CEGAR [3] can be used to reduce the abstraction.

We will need further analysis such as loop invariant for this problem, but it is outside of the thesis, and we regard it as future work.

One more observation is that CP requires a large amount of time for big size graph. This is expected since we do the fixed-point operation multiple times. Better refinement can be done to reduce the time, but currently we will not focus on such problems.

Remark 9. There are cases that CP results time-out. This is because we have set the time-out limit for each run. Most of the cases involve running the fixed point operations multiple times with the big-size graph. Running CP version again without time limit on such samples eventually finishes.

In addition, since BE-PUM and Jakstab only cover a subset of x86 assemblies, there are problems such as parsing error (missing arguments), unknown registers (bp, sp), . . . , miscalculating the next address, ... BE-PUM treats those cases as unexpected operations and stops the path.

Chapter 7

Summary and Future works

We have designed a constant propagation analysis that can be applied to X86 assembly programs, and can be done on-the-fly together with BE-PUM execution. When a path goes to an already visited location, it will only continue if the jump condition is a constant. This process is similar to loop unrolling, but we only unroll if there is a guarantee that the normal execution will also follow the same path, and since we have assumed that input programs are terminating, the unrolling process will also terminate. Experiments are done to compare the three choices:

1. `(BP)`: BE-PUM with normal unrolling (i.e. we continue exploring the path even when it goes to an already explored location)
2. `(CP)`: BE-PUM where we only unroll if the condition of a jump is a constant.
3. BE-PUM where loop unrolling is replaced by convergence check.

on 1000 real-world malwares. Since using constant propagation to unroll the loop is a *better* choice than simply skipping it, `CP` always results the better CFG. Comparing BE-PUM with and without constant propagation analysis, there are 30% of the cases that BE-PUM finishes with better results. This is expected since `CP` results an under-approximation of the CFG. We only unroll the loop if there is a guarantee that the path is taken for any initial environment.

Future works

One observation is that in our sample set, there are not much conditional branches. Due to time limit, we have not checked for this problem and regard it as future work. Another problem is the stack. Although we have restricted to using only push and pop operations, there are cases that constant propagation does not terminate (section 5.5). In the meantime we skip such problem by using the ω stack, but it needs further analysis in the future. Furthermore, our main target is the completeness of the CFG. Although `CP` eventually finishes, its result is an under-approximation of the binary program: there have the cases that the skipped paths eventually go to new locations. To generate the complete CFG, we will need further analysis such as loop invariant, refinements, . . .

Bibliography

[1] Gogul Balakrishnan, Radu Gruian, Thomas W. Reps, and Tim Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 250–254, 2005.

[2] Guillaume Bonfante, Jean-Yves Marion, and Daniel Reynaud-Plantey. A computability perspective on self-modifying programs. In *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 231–239, 2009.

[3] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000.

[4] Andreas Holzer, Johannes Kinder, and Helmut Veith. Using verification technology to specify and detect malware. In *Computer Aided Systems Theory - EUROCAST 2007, 11th International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Spain, February 12-16, 2007, Revised Selected Papers*, pages 497–504, 2007.

[5] Tomonori Izumida, Kokichi Futatsugi, and Akira Mori. A generic binary analysis method for malware. In *Advances in Information and Computer Security - 5th International Workshop on Security, IWSEC 2010, Kobe, Japan, November 22-24, 2010. Proceedings*, pages 199–216, 2010.

[6] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA 2005, Vienna, Austria, July 7-8, 2005, Proceedings*, pages 174–187, 2005.

[7] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 423–427, 2008.

[8] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, pages 214–228, 2009.

[9] Aditya V. Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas W. Reps. Directed proof generation for machine code. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 288–305, 2010.

[10] Glynn Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press, 1993.