UTS UNIVERSITY OF TECHNOLOGY SYDNEY

Static Symbolic Execution in SVF

by Guanqin ZHANG

Thesis submitted in fulfilment of the requirements for the degree of

Master of Analytics (Research) in Computer Science

under the supervision of Yulei SUI

University of Technology Sydney
Faculty of Engineering and Information Technology

Jan 2022

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, *Guanqin ZHANG* declare that this thesis, is submitted in fulfilment of the requirements for the award of *Master of Analytics (Research) in Computer Science* in the *Faculty of Engineering and Information* at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note: Signature removed prior to publication.

Date: Jan 25th, 2022

Abstract

This thesis presents the implementation of a static symbolic execution (**SSE**) tool on top of SVF framework. SSE is a well-studied technique for program testing and verification, which is based on collecting the program execution paths and exploring the program behaviours in the form of the constraints. SSE replies on SVF-IR language to translate the program into logical formulas of a particular rules-based form which supplies useful program properties for use in the program analysis. SSE maintains all elements of program running data and its manipulations. More precisely, SSE traverses on program control flow graph and modulates program behaviours, e.g., buffer-over-flow is an anomaly in program, which can be identified that the data overruns the buffer's boundary and overwrites adjacent memory locations. Experiments conducting our implementation of SSE indicates many benefits by using symbolic execution, e.g., depth-first searching on particular paths to mimic real program execution process, or finding certain program anomalies based on translating rules.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces static symbolic execution in program analysis. After giving the motivation of this dissertation, a working example is presented to illustrate the general concept of static symbolic execution and procedures. Then the research objective and contribution are summarised. Finally, the dissertation structure and our static symbolic execution framework are outlined.

1.1 Motivation

Program verification is complex but important in certifying software's reliability and security. Many software verification tasks require checking whether certain codes in the programs performed as designed. For example, an automatic analysis tool for program vulnerability detection needs to test out the existence of any weaknesses [22], such as computer viruses, trojan horses, ransomware, spyware, buffer overflows and logical error. Because of the harmful effects on our various computer systems, it is vital to research on detecting and preventing the program vulnerabilities. In addition, due to the fact that software grows rapidly in complexity and some obfuscation techniques are applied to the source code makes the traditional program verification methods impossible to process

such manually code reviewing, code testings using different and random concrete inputs, but weaknesses are only hit by specific execution conditions, which is not efficient to catch the pattern of wrong codes.

Symbolic execution [12] (**SE**) provides an efficient automated solution to these problems by exploring paths through a program. **SE** is a useful analysis technique to find the relationships between multiple inputs and outputs. **SE** exploits program states and symbolic values that reach to them. When analyzing a program, **SE** eases the programmer debugging process by supplying more semantics of the program and precise information, including program fault locations.

SE can be mainly classified into two branches: dynamic and static. Dynamic symbolic execution supplies a set of symbolic values when truly executing the program. While the instrumentation of the program cannot guarantee the ratio of the testing coverage in the program, and dynamic method cannot capture the precise fault locations. **SSE** is built on static analysis concept that reads the whole program and translates the code into logical formulas before truly executing the target program based on the designed rules, where the formulas represent the desired software semantics on path. **SSE** computes path constraints as the logical equation representations (or logical sequences) based on the instructions with the symbolic values input. **SSE** is an effective way to address automatising the program verification tasks, like bug detection. The automatic theorem prover (or constraints solver) is used to solve the path constraints to detect if the path is feasible or not. The checking results will be the modulated program execution results. In this thesis, we contribute this work on program verification in static symbolic execution.

1.2 Example

Figure 1.1 shows the basic idea of symbolic execution method. Function `foo` has two parameters including integer `x` and `y`. We assume our target is to decide which inputs

trigger the `assert` (in $line11$) fail. When we analyze the code using symbolic values for its inputs, but not the concrete value (e.g. integer number in this function), **SSE** overcomes the limitation of the generating thousands of concrete distinct values to test the function whether it can trigger the program bugs. **SSE** exploits and reasons the constraints represented by symbolic values to reason on range of typed inputs, but not concrete values of parameters.

```
1  int foo(int x, int y){
2      int a;
3      if (x > y){
4          a = 0;
5      } else {
6          if (x == y)
7              a = -1;
8          else:
9              a = 1;
10     }
11     assert(x+y != 3);
12     return a;
13 }/* Hitting the assertion as an error */
```

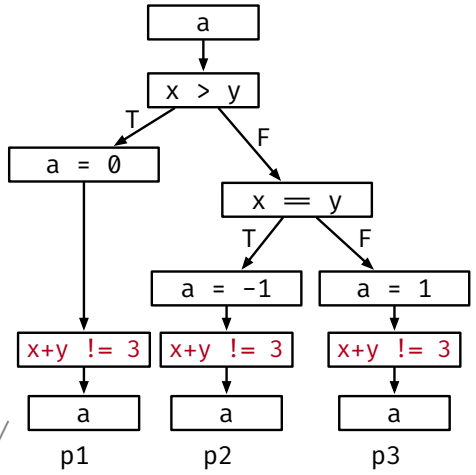

Figure 1.1: C code example and its control flow paths

To be more specific, during the symbolic execution on function `foo`, the two `if` statements cannot control the branch of the program, because x and y now are symbols, which can be any integer typed value from this context. The method is to build the logical expressions to represent the relations between x and y in each branch, below

$$\text{Path1 (p1)}: \quad (x > y) \wedge (a = 0) \wedge (x + y \; != 3)$$

$$\text{Path2 (p2)}: \quad \neg(x > y) \wedge (x == y) \wedge (a = -1) \wedge (x + y \; != 3)$$

$$\text{Path3 (p3)}: \quad \neg(x > y) \wedge \neg(x == y) \wedge (a = 1) \wedge (x + y \; != 3)$$

The powerful and detailed sequences described the importance of the x and y relations in the program. Each path sequence can be interpreted as specific path constraint and program behavior. A path constraint gives the conditions from the inputs, in the

beginning of this code x > y. When this condition is fulfilled, the execution of this path will continue going down. For example, when selecting p1, the first condition goes through true branch and ends with x + y ! = 3 assertion. The program behavior is exhibited by a = 0. Then we assume path constraint has an assignment (a = 0) under (x > y) ∧ (x + y ! = 3) constraint evaluating to True, which makes this path feasible. Otherwise, if p1 constraint for the path goes to the condition which is unsatisfied by x > y, p1 is infeasible.

1.3 Research Contribution

In this dissertation, we developed a framework for static symbolic execution which is implemented on **SVF** [19, 25, 26]. The tool will be used to assist automated program verification and understanding the program properties. To be more specific, we developed an **SSE**-tool for program analysis to solve analysis to solve the control-flow path reachability problem and to help interpreting the program properties by incorporating symbolic execution with program dependence analysis. We formulated the program representation to demonstrate our general analysis concepts and strategies. Then we present multiple examples to illustrate the experimental processing which shows the effectiveness of **SSE**-tool. In the end, we give the future work to improve **SSE**-tool's approach.

1.4 Framework Overview

Figure 1.2 describes the designed static symbolic execution overview in this thesis. In Phase A, the program is pre-processed to compile in LLVM intermediate representation (**IR**). Then in Phase B, the **IR** code are taken as the input to **SVF** and computed to build inter-procedural control flow graph (ICFG) to represent the program. Our method will explore all the desired paths from ICFG. Paths are collected by applying depth-first

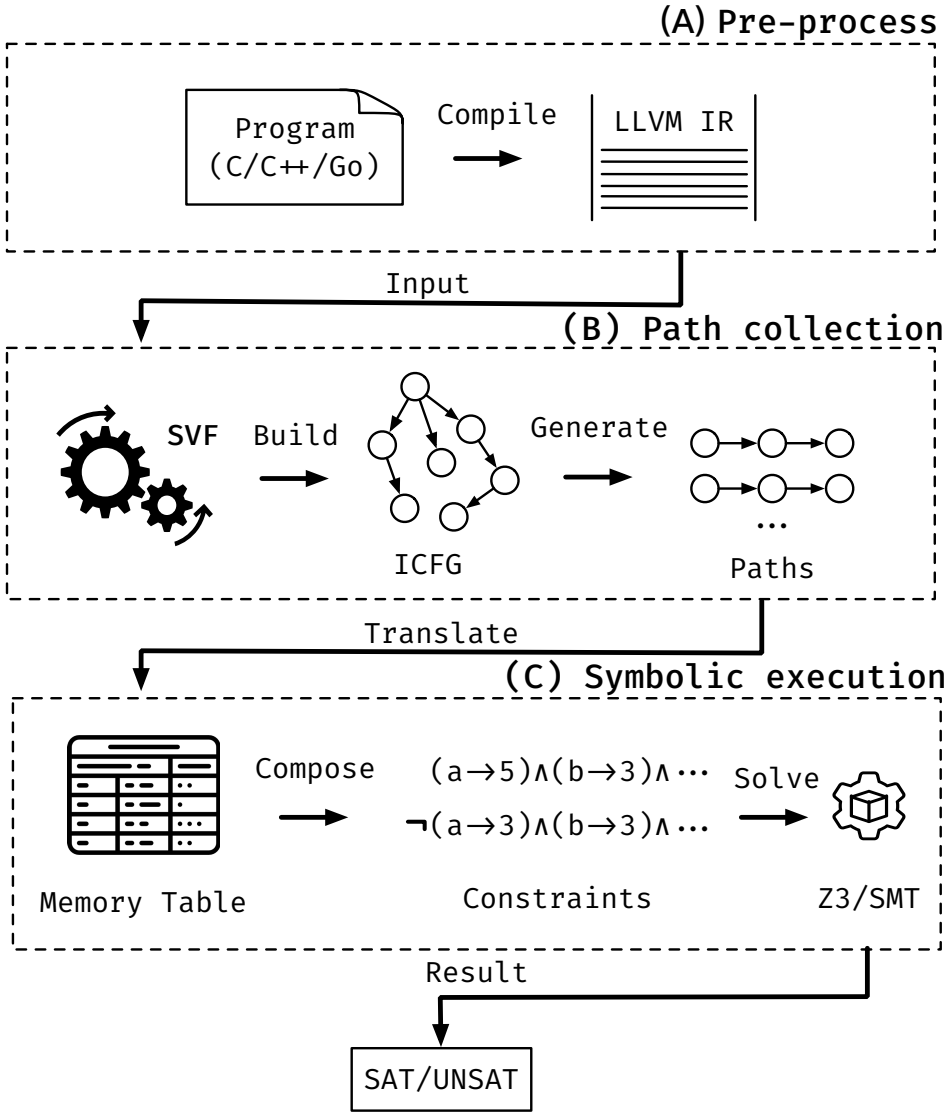

Figure 1.2: The overview of the proposed **SSE**-tool in thesis.

search algorithm. Then in Phase C, each of the path is traced from beginning to the end, and instructions will be iteratively interpreted. Each instruction will be translated to the program state stored in memory table, which are symbolic values and symbolic value's relations. After this, we will select these relations and compose program constraints in first-order logical formula form. Finally, **Z3** as one of the satisfiability modulo theories (SMT) solvers will automatically calculate the final results to check whether each path's constraints are satisfied or unsatisfied.

1.5 Thesis Structure

This thesis structure is followed as:

- Chapter 2 introduces the thesis foundations and backgrounds required in static symbolic execution. It describes program representation is explained, as well as analysis domain and its simplified LLVM-like language. Additionally, static analysis in control dependence analysis and data dependence analysis are presented.
- Chapter 3 gives the introduction of **SVF**. We formulate the definition of the ICFG by using our simplified language. Then we use graph traversal method to collect paths to be analyzed by **SSE**.
- Chapter 4 gives **SSE**'s general idea, symbolic translation rules and constraint solver are presented. Implementations are crafting as main steps - iteration of the instructions from the path - doing memory management - composing constraints formula - sending to **Z3** - final results.
- Chapter 4 evaluates the **SSE** efficiency and validation of the implementation. We demonstrate five typical examples to incrementally compute the procedures of our **SSE** methods.
- Chapter 5 summaries the thesis and indicates possible directions in the future.

Chapter 2

Preliminaries

In this chapter, we introduce theoretical backgrounds of this thesis. We firstly introduced the basic concept of symbolic execution, which is separated into control dependence analysis and data dependence analysis. Then we give the background knowledge of the LLVM-IR and introduction of SVF. Finally, we summarise program representation domain to clarify the reference of our defined simplified LLVM-like language.

2.1 Symbolic execution

Symbolic execution was introduced and implemented in EFFIGY [17] by James C King. This technique describes the data objects are represented by the variables, manipulated by statements, followed by program statement execution orders. Symbolic execution is basically defined as an extensive modulation of real execution, where its inputs are symbolic values, and it produces symbolic formulas as its outputs.

In this thesis, we developed static symbolic execution (**SSE**) followed by the basic traditional terminology [6, 7, 12]. **SSE** is precise: the analysis results are based on the each of path of error locations, effectively reducing the portion of program to be analyzed and verifying whether the vulnerable or potential high-risk sites is reachable. To automatise

these verification tasks in a more generic context, *data dependence analysis* [1, 14] and *control dependence analysis* [4] are utilised from **SVF**.

2.2 Control & Data dependence analysis

The main target of the Control dependence analysis is to determine and approximate a certain set of program instructions, functions, blocks dependencies in control relations. The fundamental concept of control dependence analysis has been studied and discussed quite extensively [2, 3, 24, 29]. Control dependence information captures the execution order effects from the statements in the code. In Subsection 3.1, we will define inter-procedural control flow graph which maintains extra abstract call stack to mimic the run-time call stack as the context sensitive information for the entire program.

Data dependence analysis [21] is our principle to describe the data objects are created and used by another operation in the program. Data dependence analysis ignores procedure calls and formulates the facts in data flow equations which are applied in different rules. The equations follow Tarski's fix-point theorem [28]. Data dependence analysis is the process of collecting information that obtains at each point in a procedure. Each procedure can be assumed as the LLVM instruction (or `SVFStmt`), which is presented in Table 2.4.2. Data dependence on an instruction if i. *stmt* initially defines a variable v, where $v \in \mathbb{V}$, and ii. `stmt` uses v.

2.3 LLVM IR

LLVM[1] is a widely used language parsing framework for program compilation, optimising, and transformation with a uniformed Intermediate Representation (IR). Our analysis targets multiple source languages programs parsed by Clang, which is LLVM compiler

[1] `https://llvm.org/`

translating code to . Many existing program analysis approaches [5, 8, 19, 20] work on the top of this popular and robust code representation. The utilization of a uniformed and formally defined language for code representation makes the relatively program analysis prominently easy to apply with new analyzing disciplines.

Construction LLVM Code representation, similar to an abstract assembly language, provides typed-based, low-level operations, and representation of 'high-level' language clearly. The designing purpose are mainly in three different scenarios: **(1)** compiler IR works in machine for in-memory language, **(2)** the representation of on-disk bitcode which is suitable for fast loading by a Just-In-Time [18] compiler, and **(3)**the representation for human to read in assembly language. The three different forms of LLVM are all equivalent. In Table 2.4.1, we describe the human readable representation domain and notation and has distinguished features:

- In LLVM-IR, variables are classified into two types: i. global variables (distinguished by prefix @) which are top level pointers. ii. the local variables are the register with % which are objects which can not be referenced.
  - Global variables are viewed as pointers in LLVM-IR, so that they are required to be explicitly dereferenced by using `load` and `store` instructions to read or rewrite, respectively.
  - Local variables have temporary and stack-taken types. Prefixed with infinite set of temporary numbers, LLVM-IR instructions return values are assigned to a local one, because temporary variable can be only assigned once.
- LLVM-IR is a strictly typed representation which hold values of primitive types (i.e. `i8` a 8-bit integer) and instruction result is typed as well.

2.4 SVF

SVF [25–27] is an open source Static Value-Flow Analysis Framework. This tool serves as a the foundation for symbolic execution in this thesis. SVF resolves both control and data dependence of a program. SVF accepts LLVM-IR and build inter-procedural control flow graph based on the LLVM-IR instructions. `SVFStmt` are the ICFGNodes, simplifying the representation of the LLVM-IR for the static analysis purpose.

2.4.1 Domain

To represent the target programs clearly, we defined our program analysis variables types and domain which are listed in Table 2.4.1.

Program Variables	Domain	Meanings
SVFVar	$\mathbb{V} = \mathbb{P} \cup \mathbb{O}$	Program Variables
ValVar	$\mathbb{P}$	Top-level variables (scalars and pointers)
ObjVar	$\mathbb{O} = \mathbb{S} \cup \mathbb{G} \cup \mathbb{H} \cup \mathbb{C}$	Memory Objects (stack, global[2], heap and constant data)
FIObjVar	$o \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H})$	A single (base) memory object
GepObjVar	$o_i \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H}) \times \mathbb{P}$	i-th subfield/element of an (aggregate) object
ConstantData	$\mathbb{C}$	Constant data (e.g., numbers and strings)
Program Statement	$l \in \mathbb{L}$	Statements labels

Table 2.1: Program Variables & Domain

- $\mathbb{P} = \mathbb{S} \cup \mathbb{G}$ represents all top-level variables, which can be split into two subsets: $\mathbb{S}$, or symbols starting with % in LLVM bit-code, which represents all stack virtual registers and $\mathbb{G}$, or symbols starting with @, which represents all global variables.
- $\mathbb{P}$ is a set of the entities whose value are explicitly stated in memory address whereas $\mathbb{O}$, a set of abstract objects corresponding to memory location, is implicitly stated, which can be only accessed indirectly by LOAD and STORE instructions through top-level pointers.

- $\mathbb{V} = \mathbb{P} \cup \mathbb{O}$ represents the union of these two set as all program variables, and our analysis is for the strong typed programs so that all variables have a domain property $\mathbb{T}$.
- $\mathbb{L}$ represents the program labels, which indicate the next instruction from control flow.

2.4.2 LLVM-like language

We unify the presentation of code examples and related instructions discussed in this thesis by using the LLVM-like language, represented by `SVFStmt`, which is conceptually as low level language to approximate the real code to omit many other trivial details and present clearly and friendly to readers, e.g. C/C++ users. This is also applied to the referred each statement as the instruction in algorithms.

SVFStmt	LLVM-Like form	C-Like form	Operand types
AddrStmt	%ptr = alloca or constantData	p = alloc or p = c	$\mathbb{P} \times (\mathbb{O} \cup \mathbb{C})$
CopyStmt	%p = bitcast %q	p = q	$\mathbb{P} \times \mathbb{P}$
LoadStmt	%p = load %q	p = *q	$\mathbb{P} \times \mathbb{P}$
StoreStmt	store %p, %q	*p = q	$\mathbb{P} \times \mathbb{P}$
GepStmt	%p = getelementptr %q, %i	p = &(q → i) or p = &q[i]	$\mathbb{P} \times \mathbb{P} \times \mathbb{P}$
PhiStmt	%p = phi [l_1, $\%q_1$], [l_2, $\%q_2$]	p = phi($l_1 : q_1$, $l_2 : q_2$)	$\mathbb{P} \times (\mathbb{L} \rightarrow \mathbb{P}^2)$
BranchStmt	br i1 %p, label $\%l_1$, label $\%l_2$	if (p) l_1 else l_2	$\mathbb{P} \times \mathbb{L}^2$
UnaryOPStmt	p = ¬q	p = ¬q	$\mathbb{P} \times \mathbb{P}$
BinaryOPStmt/CmpStmt	r = ⊗ p, q	r = p ⊗ q	$\mathbb{P} \times \mathbb{P} \times \mathbb{P}$
CallPE	%r = call f(...$\%q_i$...) f(...$\%p_i$...){ ... ret %z} $\%p_i = \%q_i$ $(1 < i < n)$	r = f(..., q_i,...) f(..., p_i,...){... return z} $p_i = q_i$ $(1 < i < n)$	$(\mathbb{P} \times \mathbb{P})^n$
RetPE	%r = %z	r = z	$\mathbb{P} \times \mathbb{P}$

⊗ ∈ {+, -, *, /, %, <<, >>, <, >, &, &&, <=,>=, ≡, ~, |, ∧ }

Table 2.2: LLVM-like language

Given p, q ∈ $\mathbb{P}$ and c ∈ $\mathbb{C}$, a small subset instructions of the relevant for the purposed of symbolic execution analysis will be used by all of our examples in this thesis.

- `AddrStmt` represents `alloca` instruction, which is used to allocate memory on stack or heap. It accepts an allocation site on memory or a constant number, and returns

a pointer. For example, `ptr = alloca i32` yields a type of integer `32-bits` (`i32`) pointer (`%ptr`).

- `CopyStmt` represents `copy/bitcast` instruction, which is to copy macro available in C/C++, whereas `q` copies the element of the value from `p`, so that the operand types are both top-level pointers.

- `LoadStmt` represents `load` instruction, which specifies the memory address from which to read. And `StoreStmt` is used to write a value to a memory address. In another word, `LoadStmt` moves the variables from memory to the registers, which are specified with $\mathbb{P}$, and `StoreStmt` moves the registers to the memory, which are specified with $\mathbb{P} \cup \mathbb{C}$ respectively.

- `GepStmt` is for the access to an address memory, which is containing a subset of derived multiple types of data. Aggregated field represents a packed data members in memory. For example, `struct` in memory may have any typed members, which can be accessed by using `load` and `store` instructions to yield a pointer to its field with the `getelementptr` instructions in LLVM-IR. And `array`'s subset elements are sequenced in memory which are declared with one underlying type, for example, `[3 × i8]` is an `array` of three elements in `8-bit` integer typed values.

- `CallPE` included direct call and indirect call, may also accept variable (include top-level pointer) as the first argument. `RetPE` represents `return` instruction will return the corresponding call instruction's specified typed value.

The above eight instructions are related to the representation of operations on memory. Data dependence analysis and its applied rules are partially based on these instructions. The following four control instructions which belong to the control relations are explained below:

- `PhiStmt` represents `phi` instruction or operation in SSA is to take the value by choosing the value from which is indicated either l_1 or l_2. And the indicated label is decided by the branch on ICFG, which will be illustrated in Subsection 3.1.

- `UnaryOpStmt` is the operating on a single operand and producing the same typed single value with its operand.

- `BranchStmt` represents `br` instruction takes single value to transfer to the indicated label-leading instructions. The branch can be classified as a conditional branch or an unconditional branch. Based on the calculation of the conditional result in `p`, if the result is evaluated to "true", control flows to the "if-true" label in LLVM-IR. If the result is "false", control instruction flows to "if-false" label respectively.

Chapter 3

Path Collection

In this chapter, we focus on collecting paths on inter-procedural control flow graph (**ICFG**), which built from SVF. Each path would be taken by SSE to perform symbolic execution as the modulation of the program manners. Expanding all paths, while hampered by paths containing branches and calls, is in higher usage of memory. The context sensitive analysis will be discussed for strategies in selecting paths. To demonstrate the consistent and simplified concept, we use our defined LLVM-like language in Table 2.4.2.

3.1 Inter-procedural control flow graph (ICFG)

Definition 3.1 (ICFG). ICFG $\langle N_{\texttt{ICFG}}, E_{\texttt{ICFG}} \rangle$ is a directed graph in which a set of $N_{\texttt{ICFG}}$ contains each node to represent each statement (`SVFStmt`) in a program, and a set of directed edges $E_{\texttt{ICFG}}$ contains each edge which related two distinguished nodes. $n_1 \rightarrow n_2$ denotes that n_1 has a direct edge to n_2. The directed edges represent the execution order or control dependent relations in the program.

We defined five different types of $N_{\texttt{ICFG}}$ and four types of $E_{\texttt{ICFG}}$ to compute and represent the program P in Table 3.1 as a graph-based program representation.

ICFG Node	LLVM Instruction	ICFG Edge	LLVM Instruction
IntraBlockNode n_{inst}^{intra}	$p = \&a$ $p = *q$ $*p = q$ $p = (t)\ q$ $p = \&(q \rightarrow fld)$ $p = \&q[c]$ $p = \&q[v]$ $p = !q$ $r = p \otimes q \mid r = p \otimes c$	IntraEdge^1 $n_{inst_1} \xrightarrow{\text{Intra}} n_{inst_2}$	$inst_1; inst_2$
		IntraEdge^2 $n_{inst} \xrightarrow[p]{\text{Intra}} n_{inst_1}$ $n_{inst} \xrightarrow[\neg p]{\text{Intra}} v_{inst_2}$	$br\ p, inst_1, inst_2$
CallBlockNode n_{inst}^{call}	$r = f(p_1, \ldots, p_n)$	CallEdge $n_{inst} \xrightarrow{\text{Intra}} n_{inst}^{ret}$ $n_{inst} \xrightarrow[inst]{\text{Call}} n_f^{entry}$	$r = f(p_1, \ldots, p_n)$ $f(p'_1, \ldots, p'_n)\{\overline{inst}\}$
RetBlockNode n_{inst}^{ret}	$\boxed{r} = f(p_1, \ldots, p_n)$		
FunEntryNode n_f^{entry}	$f(p_1, \ldots, p_n)\{\overline{inst}\}$	RetEdge $n_f^{exit} \xrightarrow[inst]{\text{Ret}} n_{inst}^{ret}$	$r = f(p_1, \ldots, p_n)$ $ret_f\ q$
FunExitNode n_f^{exit}	$ret_f\ q$		

Table 3.1: ICFG nodes and edges

- IntraBlockNode (n_{inst}^{intra}) stands for eight types of instructions including Allocation, Load, store, cast, field, array, unary and binary.
- CallBlockNode (n_{inst}^{call}) represents call instruction which directly connects to the program callee's function entry.
- FunctionEntryNode (n_f^{entry}) is the starting point of the function block. n_f^{entry} contains a set of formal parameters from the caller functions.
- FunctionExitkNode (n_f^{exit}) represents the function exit instruction.
- ReturnBlockNode (n_{inst}^{ret}) represents instruction in function return value.

As we mentioned in Section 2.2, control dependence can be viewed as the execution of one instruction controls whether another is executed. And **ICFG** supplies the direct control dependencies, even two nodes on **ICFG** are logically non-related, and could be executed in either order without different results. Our definition of four types of edges summarised all five types of nodes relations with each other, below:

- INTRAEDGE[1] represents a definite order of the control relation which is sequentially executed begin with one instruction to another.
- INTRAEDGE[2] represents the branch instruction, and the execution of the next instruction is depending on the condition's result. For example, the "$if-else$" conditions branch in program, the program only goes to one branch.
- CALLEDGE stands for the nodes to invoke caller function or returned valued by the invoked the function. It holds the instruction with its distinct call-site id.
- RETEDGE stands for instruction goes to the function's exit, which is as the callee to hold the identical call-site id matching to its caller function.

3.2 ICFG Path

SSE handles each path separately, and translates each instruction along the path into the path-based constraints. And each node on **ICFG** is a `SVFStmt`, which can be viewed as a program operation state. SSE propagates states along the path, and requests constraint solver to justify whether the current path is feasible or infeasible. However, counting and modulating on all paths of a program is expensive. In many software engineering testing and debugging strategies [7], they conduct selective paths collection to find the most effective and representative paths when optimising the whole process. KLEE [9] makes the heuristics path selection process, which is aimed help symbolic execution fulfill the

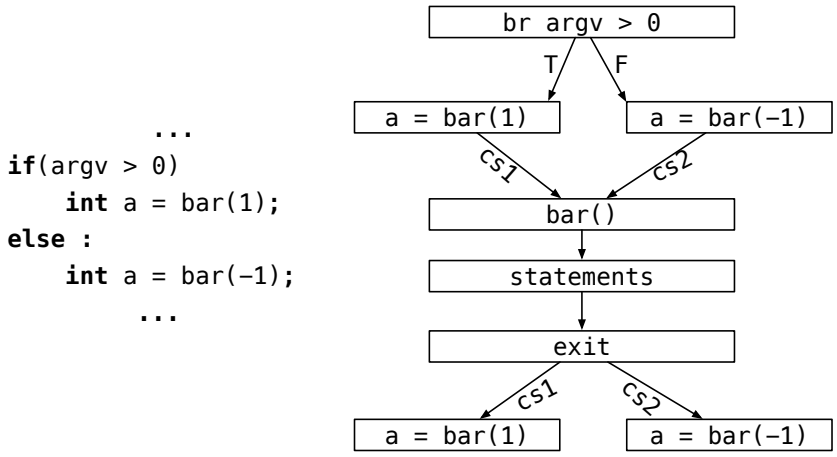

Figure 3.1: Context Sensitive Case Example

specifically requested clients (e.g., memory leak detection, stack-over flow detection, etc). The most promising and optimal strategy is still an open problem in symbolic execution domain.

Depth-first search (**DFS**) is implemented for graph analysis algorithm, which explores the paths along to the deep subnodes before doing backtrack, while breath-first search (BFS) expands the path more diversely. With the path going deeper, the number of paths is increasing exponentially. We applied context sensitive path collection in modeling program states as real program execution, while our definition of the **ICFG** is on *intra*-procedural level, where all paths through a program are built on graph in our defined LLVM-like language. When one function is called in multiple times, the program states are varied in different context, which is defined context sensitive analysis.

Context Sensitive Analysis In Figure 3.1, code fragment shows the context sensitive example and its **ICFG**. The condition $'argv > 0'$ depends which value will be passed to $bar()$ function. However, no matter in which condition, $bar()$ function is invoked, we can't simply traverse and collect all paths, because when $argv > 0$ is fulfilled and $bar(1)$ is called, the control flow path to $bar(-1)$ is invalid. On the contrary, when $argv > 0$ is not fulfilled, the control flow path to $bar(-1)$ is valid. We uses CALLEDGE and RETEDGE

which are denoted with distinct call-site id. So that we can match the current invoked call call-site (cs) with return call-site to ensure the context sensitive path.

The recursive algorithm in Algorithm 1 marks the visited node in the $visited$ which is a map of visited ICFGEDGE to cs. And cs is a stack to collect the call instructions' call-site, applied with $first-in$ and $last-out$ in order to retrieve the matching calling context which is on the context sensitive analysis. Shortly, we will discuss the context sensitive and path sensitive in the next two paragraphs. And the result is π which is a sequence of the edges on **ICFG**.

Algorithm 1 Depth-first Traversal on ICFG

Input: $\texttt{edge}_{\texttt{src}} \in E_{\texttt{ICFG}}$, $\texttt{dst} \in N_{\texttt{ICFG}}$

Output: All possible paths from $\texttt{n}_{\texttt{src}}$ to $\texttt{n}_{\texttt{dst}}$ on ICFG

Initialize:
  $cs = \{\}_{stack}$: a sequence of call stack
  $\pi = \{\}_{array}$: a sequence of edges in the current traversal path
  $\texttt{visited} = \{\}_{\texttt{map}}$: a map of visited edges to $cs = \{\}_{stack}$
function $DFS(\texttt{curEdge}, \texttt{dst})$:
  $\texttt{curItem} \leftarrow \langle \texttt{curEdge}, \texttt{callstack} \rangle$
  visited.insert(curItem)
  π.push_back(curEdge)
  if $\texttt{edge}_{\texttt{src}}.\texttt{dst} == \texttt{dst}$ **then**
    $paths.insert(\pi)$
  for each $\texttt{edge}_{\texttt{src}}.\texttt{dst} \rightarrow \texttt{n}'$ **do**
    if $\texttt{edge}_{\texttt{src}}.\texttt{dst} \rightarrow \texttt{n}' \notin visited$ **then**
      if $\texttt{edge}_{\texttt{src}}.\texttt{dst} \xrightarrow{\textsc{Intra}} \texttt{n}'$ **then**
        $DFS(\texttt{edge}_{\texttt{src}}.\texttt{dst} \rightarrow \texttt{n}', \texttt{dst})$
      else if $\texttt{edge}_{\texttt{src}}.\texttt{dst} \xrightarrow[\texttt{inst}]{\textsc{Call}} \texttt{n}_{\texttt{f}}^{\texttt{entry}}$ **then**
        cs.push_back($\texttt{n}_{\texttt{f}}^{\texttt{entry}}$);
        $DFS(\texttt{edge}_{\texttt{src}}.\texttt{dst} \rightarrow \texttt{n}_{\texttt{f}}^{\texttt{entry}}, \texttt{dst})$
      else if $\texttt{edge}_{\texttt{src}}.\texttt{dst} \xrightarrow[\texttt{inst}]{\textsc{Ret}} \texttt{n}_{\texttt{inst}}^{\texttt{ret}}$ **then**
        if $\texttt{cs} \neq \varnothing$ && $\texttt{cs.back}() == \texttt{n}_{\texttt{inst}}^{\texttt{ret}}$ **then**
          cs.pop_back()
          $DFS(\texttt{edge}_{\texttt{src}}.\texttt{dst} \rightarrow \texttt{n}_{\texttt{inst}}^{\texttt{ret}}, \texttt{dst})$
        else if $\texttt{cs} == \varnothing$ **then**
          $DFS(\texttt{edge}_{\texttt{src}}.\texttt{dst} \rightarrow \texttt{n}_{\texttt{inst}}^{\texttt{ret}}, \texttt{dst})$
  visited.erase(curItem)
  π.pop_back()

Depth-first Traversal on ICFG maintains the context information on **ICFG**, and Algorithm 1 is initialized with three variables widely representing a sequenece of call stack (cs), a sequence of edges as a path (π), and a hashed visited edges to its calling context (`curItem`). SSE firstly starts from the first node of $\texttt{edge}_{\texttt{src}}$. In each step, the algorithm picks one destination node of non-visited edge from $\texttt{edge}_{\texttt{src}}$ to neighbour node n'. Then it performs the recursive call for every INTRA edge. In sorting the calling context information, call stack (`cs`) will be pushed and maintained in processing with the CALL edge. When it comes to the RET edge, `cs` last element will be matched with the return site.

As it keeps going deeply as much as possible to reach the end, where the edge's destination node doesn't have any neighbours reaching to, or it reaches to the $\texttt{n}_{\texttt{dst}}$. Then it goes to other adjacents of the node from $\texttt{n} \rightarrow \texttt{n}'$. To be noted, from $line11$ to $line17$ in Algorithm 1 is to cooperate with the context sensitive analysis, during the traversal, if the current edge is a CALLEDGE or RETEDGE, the callsite would be collected to the stack, with $first-in-last-out$ policy. When collecting CALLEDGE, the call-site is collected, and when collecting the RETEDGE, the call-site would be matched with the last pushed call-site respectively. Only matched RETEDGE can continue the current process and the matched call-site would be popped from the call-site stack. Once successfully reached $\texttt{n}_{\texttt{dst}}$, the path π then is stored into $paths$, which will be in solving process on for the symbolic execution in Section 4

Chapter 4

Symbolic execution

In this chapter, we firstly introduce the $state-of-arts$ open-source symbolic execution projects. Then we present our implementations of SSE, which describes the process of SSE in each procedure, building on the preceding defined domain and program representations foundations in Table 2.4.1 and Table 2.4.2. Encoding the control flow path into logical formulas according to the program states varies on types of `ICFGEdges`. The memory management will be illustrated during the formation of program logical formulas period.

4.1 Introduction

Symbolic execution is a concurrent technique which evaluates the program in the symbolic manners, without the testing and verifying the program with the concrete values as program inputs. Symbolic execution [6, 11, 12] has been introduced over 30 years as a concept of targeting to check the program and test the properties of the functions, code blocks, application interface and the whole project.

In general, symbolic execution emerged as the autonomous, systemic examine engine to scan and trace the certain paths of the software, as while as justify whether common

but dangerous flaw codes cause the whole program crash (e.g. unawareness of the zero divided issue, extra unreleased NULL pointers referenced problem, or even SQL injection triggered to 'sudoer'.) To address the following problems, there is one stylish solution: technologically run and trigger the software in which conditions evoke crash or fail. In stead of supplying the exact and limited test-issue input randomly, symbolic execution takes the advantage of the satisfiable constraints and scalable testing approaches. It used the automated software testing to search and explore possible logical program paths and came out with the concrete inputs as the validation of each logical path.

From the testing side, when exploring the paths in the program, symbolic execution would examine and validate the current assertions, memory leak problems, bad exceptions, and syntax errors. After that, with the full-coverage-path symbolic formulas, the logger could report the testers that in which bugs or exceptions has been occurred with the 'conditional input' triggered to happen.

4.2 Existing symbolic execution tools

Modern symbolic execution tools exploit wide range of techniques to optimize and approximate the program more precise and efficient. **DART** [13], directional automated random testing, is the first proposes tools, which combined random testing with model block checking to perform completely autonomously. The goal of DART is to straight-force execute all feasible paths of the program for checking the variables about the memory leak, NULL pointers .etc. It is the basic execution automated method to conduct software and project test.

EXE [10] is one completely symbolic execution tool built for C-language project. When EXE testing the project, it traces through one single path to reason about all possible constraints matching to the input automated testing suits. Whereas the use of the Simple Theorem Prover(STP), which is the initial period of the reasoning tools to check the

generated concrete testing cases satisfied or not. However, in terms of short-comings, EXE is limited in solving the unbounded recursive loops or call-functions. Therefore, the properties of the projects may not be totally reached by the theorem prover.

KLEE [9], is redesigned from the EXE, KLEE currently has the high-coverage of the target testing cases. The basic method of KLEE is removing the unrelated un-constraints and tracking all registered memory locations as KLEE is implemented the LLVM. KLEE then have a good performance in system bug finding and monitoring, because the released tool KLEE had the third-parties handler for reading outside environments.

4.3 An Overview of SSE

As we have introduced the motivation of symbolic execution in Section 1.1 and Section 4.1, presented the general potential tasks to be addressed by SSE. We understand static symbolic execution as a method to interpret and represent programs which contains symbolic values or symbolic value composed equations, where each symbolic value is defined by the different symbol which can be represented a set of typed concrete values, ranged over. For instance, x and y, in Section 1.2, can be ranged over all integer numbers. To understand the program behaviors and ease to represent them, we assume the program variables are storing with symbolic values, and the program statements (or instruction) can be translated into mathematical equations or logical conjunctions. The example case of source code in Figure 1.1 is translated into three paths where each path is composed logical conjunctions.

SSE memory model maintains the program states when performing static symbolic execution analysis. And the states is, a mapping from program variables' name to value, including symbolic expressions. The expressions can be a symbolic variable's constraints (e.g. $x > 10$) or a relational conjunctures (e.g. $x > y$ or $(x > 10) \wedge (y < 4)$). Our **SSE** analyzes the program statically which is based on exploring the paths including

exploring branches to gain information of each branch which we have discussed in Section 3.2. The representation of the path now is translated to the constraints, which will be put to the constraint solver to calculate the final results. A constraint solver in general is served as a black box, given a set of constraints over symbolic variables, to find a solution of all program variables that are satisfied the constraints. We will discuss this part in Section 4.6.

To summarise, SSE, with the simplest way to describe in above, is exploring all possible paths of the program, and then symbolic values are deployed to retain the complete information on paths to represent program behaviors, which can be reasoning for the purposes of program analysis.

4.4 Memory Management

Based on the existing definitions of program execution state [15, 16], all program execution starts from one of the initial states. An initial state contains mappings from the program abstract objects to symbolic values in `LocMap`, which is denoted according to the program we are trying to interpret. Our working example in Figure 1.1 is only simplified model where the data in scalar concept. The nature of symbolic state is how program execution is translated and modeled in memory. SSE performs memory operation by mapping not only the variables, but also their memory objects to the symbolic expressions, which are corresponding to our defined program variables in Table 2.4.1. In general, a referenced memory object address is allocated as a mapping which relates the memory address index with its symbolic expression.

Program execution is a transformation of memory states. Based on our simplified LLVM-like language, memory management is the constraint to translate the instructions into the modeled program states. `LocMap` is symbols to $expr$, and the input of the path sensitive solver is a sequence of **ICFG** edges, which is collocted in Algorithm 1.

The $expr$ is the symbolic expression and the syntax is below:

$Expr$	::=	Object | Int | Float | Bool | String	
	|	$Expr(v)$	Program variable v' expression
	|	$Expr(v_1) \otimes Expr(v_2)$	Binary operations $(*, +, -, /, \ldots)$
	|	$\neg Expr$	negative flags
	|	$Expr \wedge Expr$	Conjunction
	|	$Expr \vee Expr$	Disjunction

Table 4.1: Symbolic execution $Expr$ syntax

$Expr$ can be assumed as a function to generate an expression which could be the objects, integral value, floating point value, Boolean value, and string. $Expr(v)$ is the variable v mapping to the M. Each expression can be calculated with another or making it negative. The conjectures between expressions will be explained shortly.

Static single assignment is on for the definition of characteristics of LLVM-IR, about which each variable is defined once, or assumed as the objects could only be accessed by LOAD/STORE. The **ICFG** path will be translated into a set of logical constraints in the different rules based on our defined LLVM-like language instructions, which is exetended in Table 4.2.

Left hand operand: `expr p =` $Expr("p")$ Right hand operand:`expr q =` $Expr("q")$ `expr r =` $Expr("r")$
`LocMap`: a map of ValueExpr to a constant data or a location value `sol`: symbolic solver

SVFStmt	C-Like form	Operations
`AddrStmt` (constant)	`p = c`	`sol.add(p == c)`
`AddrStmt` (mem allocation)	`p = alloc`	`sol.add(p == getMemObjAddress("alloc"))`
`CopyStmt`	`p = q`	`sol.add(p == q)`
`LoadStmt`	`p = *q`	`sol.add(p == LocMap[q]);`
`StoreStmt`	`*p = q`	`LocMap[p]= q`
`GepStmt`	`p = &(q → i)` or `p = &q[i]`	`sol.add(p == LocMap[q+i])`
`PhiStmt`	`r = phi(`l_1`:p,` l_2`:q)`	`if(executed from` l_1`)` `sol.add(r == p)` `if(executed from` l_2`)` `sol.add(r == q)`
`BranchStmt`	`if (p)` l_1 `else` l_2	`expr cond = p` `if(cond.is_false())` execute l_2 `else` execute l_1
`UnaryOPStmt`	`¬r = p`	`sol.add(r == ¬p)`
`BinaryOPStmt`	`r = p ⊗ q`	`sol.add(r == (p ⊗ q))`
`CmpStmt`	`r = p ⊙ q`	`sol.add(r == ite(p ⊙ q, true, false))`
`CallPE/RetPE`	`r = f(...,q,...)` `f(...,p,...){... return z}`	
`CallPE`	`p = q`	`sol.add(p == q)`
`RetPE`	`p = r`	`sol.add(p == r)`

Table 4.2: Translation Rules

- `AddrStmt` is symbolic variables to allocation memory. Manipulations for allocation operation can be separated into two types: constant and address objects, which are the initialization of the objects.

  `getMemObjAddress("alloc")` is on for an object initialization in `LocMap`. The bit-wise operation with &, ¬, >>,<< can be used to represent the offset and its arithmetic operation in $Expr$. Initialization process is applied with `SVFVar` so that the defined objects have been allocated with a unique id in SVF.

- `CopyStmt` is simply copy the same expression to the copied one.

- `LoadStmt` is to access the `LocMap` for modeling reading operation on object. And only the context in which `p` is a program pointer can read the `LocMap(q)` inferred expression.

- `StoreStmt` is to to access the `LocMap` for modeling writing operation in memory states. And only the context in which `q` is a program pointer can be written to the expression from inferred value (`LocMap[p]`).

- `GepStmt` is on for the aggregation that indicates a structure or a class typed object. Our memory table uses allocation site with the weights correspond to the field indices. So that $Expr(p)$ is the allocation site of $Expr(q)$ added with the offset field object. So that ARRAY has two scenarios: variables or constant indices, which are both taken from '**getelementptr**', we exploit similar way from LLVM-IR retain the value from the array pointer.

- `PhiStmt` instruction's label decides in which line to the edge of l_1, $Expr(r)$ should copy $Expr(p)$, or $Expr(q)$ from l_2 respectively..

- `BinaryOPStmt` is given the opcode as the operator and two operands, which are both translated to $Expr$ then being calculated and assigned the result to `r`.

- `CmpStmt` can be assumed as the $'fork'$, $Expr(p)$ value is depending on the operations with $Expr(q)$ and $Expr(r)$, which is applied the calculation rules in Table 4.1. Rather than added into solver, we use `z3::ite` to encode both possible comparing outcomes into expression, though the final calculated results should be true or false, e.g., in `r = ite(p > 4, true, false)`, `r` finally has a distinct value true if $Expr("p") > 4$.

- `CallPE` call instruction invokes with the caller and pass to the formal parameters. For each actual parameter variable needs to be copied to each formal parameter, and the operation is similar to CAST operation. In function exit instruction, the return value with $Expr("q")$ should be copied to the invoked instruction's variable, respectively.

- `RetPE` return statements retrieve the function return value and copy the variable to the return value `r`.

SSE memory model is core design for symbolic execution in constriant solving. In the following rules and the definition of the $Expr$, we use $Expr("p")$ to represent the variable `p`'s symbolic expression, and the allocated memory objects expressions, we define `LocMap` to represent and store the expressions in &p, where &p is the base object address of the variable `p`. And for aggregated structures, e.g., `q` is an array in the `GepStmt`, and `i` is an integer constant index, by which means, `q[c]` is `&(q+c)` in `LocMap`.

4.5 Path Solving

Symbolic execution is a reliable method to solve program line reachability (or program control flow reachability [23]). Our static symbolic execution is based on exploring and collecting **ICFG** paths, then constructing memory model during the traversal from each path. Memory model will be composed to the mathematical constraints to represent

each path execution behavior, or program states. Our method is possible to prove the reachability of the control flow paths with the data dependencies. This method has two advances: $i.$ proving the existence of bugs like $logic-errors$, $buffer-overflow$, etc. $ii.$ checking certain bugs when countering to non-reachable path errors. To be more specific, when **SSE** finds an unsatisfied (or contains conflicts in path constraints) in the states, this path will be discarded and stopped continuing to translate and solve.

Constraint solving techniques are not the focus of our research. However, the application of constraint solver functionalities have significant impact on the performance of symbolic execution. We briefly introduce constraint solving in Section 4.6.

Definition 4.1 ($first-order-logic$ (FOL))**.** FOL consists of variables, (non)logical symbols which are constructing the a sentence or a formula. We defined ϕ as a formula (e.g. $\phi = (a \wedge\ b) \wedge (\neg\ c)$). And the literals of $a, b\ or\ c$ can be assumed as a variable or an expression with an assigned value.

As our **SSE** follows a path on **ICFG** through the code in different context, it collects the assignments value as a constraint equation and adds to the solver, which is explained in Section 4.4. The constraint solver (`sol`) collects where the constraints imposed with the variables' operations, e.g., $Expr("p")$ as the left hand operand in Table 4.2. The resulting process is encountering a BRANCH or a ASSERTION. The solver adds the constraints along to the rest of collected nodes which can be assumed as an incrementally solving. SMT can maintain the incremental results in the query and reuse the constraints as previous knowledge when building up the next incremental elements of the constraints.

In order to reduce the calculating complexity. In our design, we only do SMT calculation at branch condition and program assertion. At the beginning of the analysis, possible combinations of all branches and get the constraints of each combination are iterated. The SSA helps to make the variables more accurate in memory map. Therefore, in the branch condition, we form the description of the problem through the branch

Algorithm 2 Translate a control-flow path

Input:

1: π: an ICFG path;
2: `sol`: Z3 solver.

Output:

3: `SAT`: satisfied results `UNSAT`: unsatisfied results `UNKNOWN`: unsolved

```
4: function translatePath(π):
5:     for each edge ∈ π do
6:         if edge ∈ IntraEdge[1] then
7:             Obey operations in Table 4.2
8:         else if edge ∈ IntraEdge[2] then
9:             cond ← edge.getCond()
10:            successorVal ← edge.getSuccessor()
11:            res ← Eval(cond == successorVal)
12:            if res.is_false() then
13:                return false
14:            else if res.is_true() then
15:                sol.add(cond == successorVal)
16:        else if edge ∈ CallEdge then
17:            sol.push();
18:            for each callPE ∈ edge.getCallPEs() do
19:                lhs ← getZ3Expr(callPE.getLHSVarID());
20:                rhs ← getZ3Expr(callPE.getRHSVarID());
21:                sol.add(lhs == rhs);
22:        else if edge ∈ RetEdge then
23:            if retPE ← edge.getRetPE() then
24:                rhs ← getEvalExpr(getZ3Expr(retPE.getRHSVarID()));
25:                sol.pop();
26:                lhs ← getZ3Expr(retPE.getLHSVarID());
27:                sol.add(lhs == rhs);
28:            else
29:                sol.pop();
30:    return sol.check()
```

conditions and the objects stored in the map associated with the branch conditions. It has three answers, SAT, UNSAT and UNKNOWN. SAT usually means that the previously recorded variable relationship can support the current branch combination.

The path solving process is the idea of constraints rules combination from memory table. As shown in Algorithm 2, the path conditioned edge (IntraEdge[2]) has its cor-

responding condition expression in solver, stored the involved variables based on the syntax of $Expr$. So that each constraint in involved each variable assignment in solver, which are checked against the path condition with a context sensitive analysis. As to solve the calling context, i when SSE solves the CALLEDGE, SSE uses `sol.push()` to maintain the function calling scope in its calling context, and the solver also has all corresponding expression variables, ii when SSE counters to the RETEDGE, the solver will use `sol.pop()` to delete all temporal expressions, which are added to the solver after solving its corresponding CALLEDGE.

If the analysis continues, the current SMT condition constraints will be cleared. UNSAT means that the previously recorded variable relationship cannot make the program continue to execute in the current branch combination, that is, this is an impossible path and there is no need to continue analysis. UNKNOWN indicates that there are some undefined variables, which makes it difficult to determine whether to continue. In this case, the current SMT condition constraints would be retained to continue the analysis, and participate in the next constraint solution until the end of this path.

4.6 Satisfiability modulo theories

Constraint solvers are decision-making tools for problems represented by logical sequences: for example, Boolean satisfiability problems (also known as SAT) are designed to determine whether there is an interpretation of formula symbols that make them true.

SMT represents satisfiability modulo theories. The concept of Satisfiability comes from mathematical logic. If the equation or formula can be made true by selecting an appropriate value for the variable, the equation or formula is satisfiable. SMT provides more expressive language and more natural description. SMT generalizes the SAT

problem with support theory to obtain formulas involving linear arithmetic and array operations. The SMT solver maps the atoms in the SMT formula to new Boolean variables: the SAT decision process checks the satisfiability of the rewritten formula, and the theoretical solver checks the model generated by the SAT process [7].

The power of SMT comes from its ability to deal with different theories. In addition to arithmetic operations, the SMT solver can automatically infer Boolean operators (such as and, or), arrays and matrices, digital circuits, strings, and software data structures (such as lists and trees).

At the most abstract level, the language of SMT is the language of mathematical logic. Arithmetic, Boolean reasoning. Functions, bit vectors and arrays, quantifiers. We used some coding tricks here. The first is a method called static single allocation (SSA). SSA uses different names to represent the same variables at different times. Another technique is memory mapping. We abstract each object and exist a map to decouple the programming language from the specific SMT language.

Chapter 5

Validation

Our **SSE** tool is an experimental implementation, which is not necessarily and readily confronting large programs. Nevertheless, we present the multiple examples in this chapter, which is written in basic C/C++ language programs with assignments, branches, and inter-procedural function calls. These examples are enough complex to validate and describe our tool. **SSE** tool is supporting types including integers, floats and array. These examples are compiled to LLVM-like language, which is defined by us in Chapter 2. We use Z3 as the constraint solver to perform analysis, upon which is applying in Table 4.2 and Algorithm 2. Incremental solving is illustrated in each example.

5.1 Testing

Figure 5.1 shows the integer operations and compare example's source code and its **ICFG**. As we can see from the Table 5.5, two variables are initialized, where we use the same variable names as the symbolic expression name (e.g., $Expr("a")$ stands for variable `a` symbolic expression). As a result, two integer type symbolic expression has also been initialized, with the `AddrStmt` corresponding rules. Then along with the traversal on **ICFG** path, we applied with the Algorithms 4.2, until to the `assert`, we put the on going

```
1 void main(){
2     int a;
3     int b;
4     a = 0;
5     b = a + 1;
6     assert(b > 0);
7 }
```

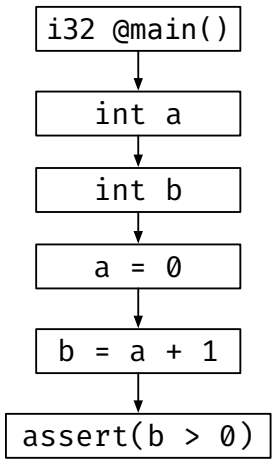

Figure 5.1: Example (1) Integer computes

lively symbolic expressions to the solver to help us reason result (`true/false`), which indicates the current path is feasible (SAT) or infeasible (UNSAT).

ICFGNode	**SVFStmt**	**Operation**	**Constraints**	**Reasoning**
i32@main()	-	-	{ }	true
int a	AddrStmt	$Expr("a")$	$\{a \rightarrow$ a$\}$	true
int b	AddrStmt	$Expr("b")$	$\{a \rightarrow$ a, $b \rightarrow$ b$\}$	true
a = 0	StoreStmt	a==0	$\{a \rightarrow$ a, $b \rightarrow$ b, a$\rightarrow 0$ }	true
b = a + 1	BinaryOPStmt	b==(a+1)	$\{a \rightarrow$ a, $b \rightarrow$ b, a$\rightarrow 0$, b$\rightarrow$ a$+1$ }	true
@assert(b > 0)	callPE for Solve	res==ite(b>0,true,false)	$\{a \rightarrow$ a, $b \rightarrow$ b, a$\rightarrow 0$, b$\rightarrow$ a$+1$ res $\rightarrow$ true$\}$	SAT

Table 5.1: Example (1) SSE Translation

Figure 5.2 shows the example with one-level integer typed pointer $*p$ used in the program. In the beginning, similarly to Example 5.5, the **SSE** state was noted with `a, p, q, b` initialization process. Then $Expr("a")$'s object is stored into the integer pointer $*p$. In memory, `LocMap[p]` is changed to $Expr("a")$. When $*p = 0$ in $line$ 5 was executed, the operation is `LocMap[p] = 0`, by which means `LocMap[p]`'s object points to a concrete

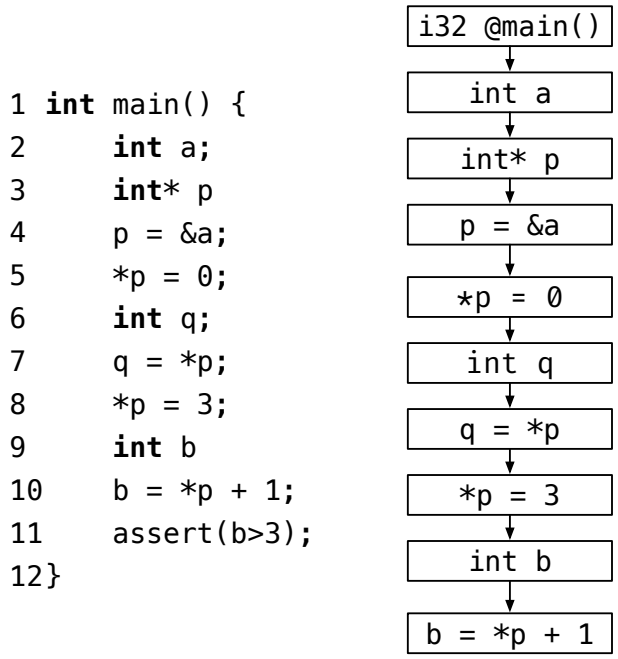

Figure 5.2: Example (2) One-level pointers

value 0. Then $Expr("q")$ is loaded by `LocMap[p]`. Next we store `LocMap[p]` with value 3. $Expr("b")$ is added by `LocMap[p]` + 1 so that `b==4`. Finally, `b>3` is asserted. Solver can look up the result of the compared value in solver and the reasoning result is SAT.

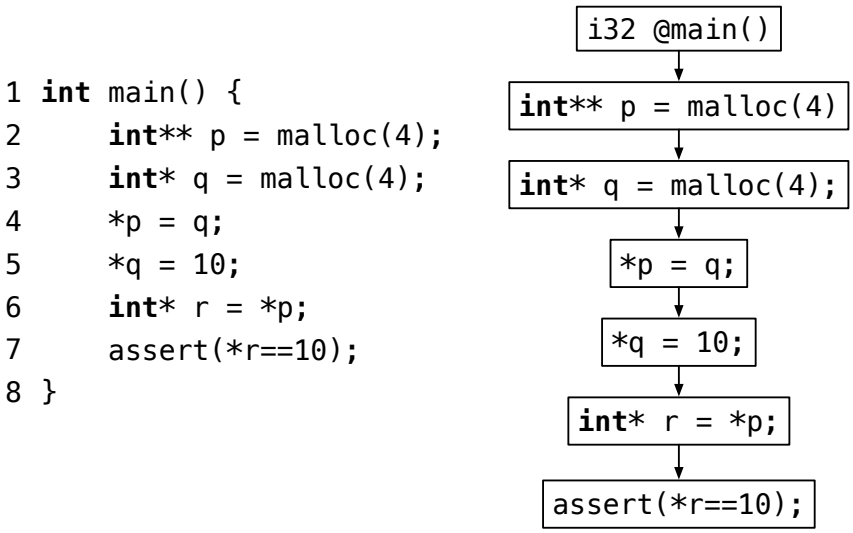

Figure 5.3: Example (3) Multiple-level pointers

Figure 5.3 presents multiple-level pointers example whereas we initialized $**p$ and

ICFGNode	SVFStmt	Operation	Constraints	Reasoning
i32@main()	-	-	{ }	true
int a	AddrStmt	$Expr("a")$	{$a \rightarrow$ a}	true
int * p	AddrStmt	$Expr("p")$	{$a \rightarrow$ a, $p \rightarrow$ p}	true
p = &a	AddrStmt	p=getMemObjAddress(a)	{$a \rightarrow$ a, $p \rightarrow$ p a$\rightarrow$ o_1}	true
*p = 0	StoreStmt	LocMap[p] = 0	{$a \rightarrow$ a, $p \rightarrow$ p a$\rightarrow$ o_1}	true
int q	AddrStmt	$Expr("q")$	{$a \rightarrow$ a, $p \rightarrow$ p a$\rightarrow$ o_1, $q \rightarrow$q }	true
q = *p	LoadStmt	q==LocMap[p]	{$a \rightarrow$ a, $p \rightarrow$ p a$\rightarrow$ o_1, $q \rightarrow$q, q$\rightarrow$ 0 }	true
*p = 3	StoreStmt	LocMap[p]=3	{$a \rightarrow$ a, $p \rightarrow$ p a$\rightarrow$ o_1, $q \rightarrow$q, q$\rightarrow$ 0 }	true
int b	AddrStmt	$Expr("b")$	{$a \rightarrow$ a, $p \rightarrow$ p, a$\rightarrow$ o_1, $q \rightarrow$q, q$\rightarrow$ 0, $b \rightarrow$b }	true
b = *p + 1	BinaryOPStmt	b==LocMap[p]+1	{$a \rightarrow$ a, $p \rightarrow$ p, a$\rightarrow$ o_1, $q \rightarrow$q, q$\rightarrow$ 0, $b \rightarrow$b, b$\rightarrow$ 4 }	true
@assert(b > 3)	Call for Solve	res=ite(4>3,true,false)	{$a \rightarrow$ a, $p \rightarrow$ p, a$\rightarrow$ o_1, $q \rightarrow$q, q$\rightarrow$ 0, ,$b \rightarrow$b, b$\rightarrow$ 4, res$\rightarrow$true }	SAT

Table 5.2: Example (2) SSE States Transitions

$*q$ with $malloc(4)$ to allocate the requested memory. And as we mentioned, variable's pointers are all initialized, so that $*r$ is also finished in the beginning. Similarly, in $line4$ and $line5$, $*p = q$, $*q = 10$ are StoreStmt, so that we use LocMap referenced expression to store the variable and value for q and constant data 10, respectively. Last but not least, $*r = *p$ in the combined operation, expression for r in LocMap is stored, then load the data from $*p$, by which get the expression from LocMap[p].

Figure 5.4 gives the example of structure data type. Firstly, the referenced pointer $*p$ is initialized and stored with in the memory. Particularly, in $line9$ and $line11$, one level pointer and two level pointer are stored with the offset data from p, so that the $Expr("q")$

ICFGNode	SVFStmt	Operation	Constraints	Reasoning
i32@main()	-	-	{ }	true
int** p = malloc(4)	AddrStmt	$Expr("p")$ p=getMemObjAddress(malloc1)	$\{p \rightarrow p, p \rightarrow o_1\}$	true
int* q = malloc(4)	AddrStmt	$Expr("q")$ q=getMemObjAddress(malloc2)	$\{p \rightarrow p, p \rightarrow o_1$ $q \rightarrow q, q \rightarrow o_2\}$	true
*p = q	StoreStmt	LocMap[p]=q	$\{p \rightarrow p, p \rightarrow o_1$ $q \rightarrow q, q \rightarrow o_2\}$	true
*q = 10	StoreStmt	LocMap[q]=10	$\{p \rightarrow p, p \rightarrow o_1$ $q \rightarrow q, q \rightarrow o_2\}$	true
int* r = *p	StoreStmt	$Expr("r")$ r == LocMap[p]	$\{p \rightarrow p, p \rightarrow o_1$ $q \rightarrow q, q \rightarrow o_2$ r$\rightarrow$ 10}	true
assert(*r == 10)	Call for Solve	res=ite(10==10,true,false)	$\{p \rightarrow p, p \rightarrow o_1$ $q \rightarrow q, q \rightarrow o_2$ r$\rightarrow$ 10, res$\rightarrow$ true}	SAT

Table 5.3: Example (3) SSE States Transitions

```
1  struct A{
2      int f0; int* f1;
3  };
4  int main() {
5      struct A* p = malloc;
6      int a;
7      int* x = &a;
8      *x = 5;
9      int* q = &(p->f0);
10     *q = 10;
11     int** r = &(p->f1);
12     *r = x;
13     int* y = *r;
14     int z = *q + *y;
15     assert(z==15);
16 }
```

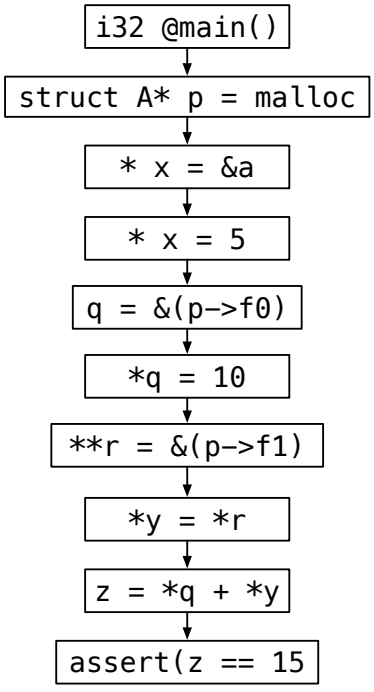

Figure 5.4: Example (4) Structure

and $Expr("r")$ should be stored with the object from LocMap which are initialize from the top level pointer with its offset respectively.

ICFGNode	SVFStmt	Operation	Constraints	Reasoning
i32@main()	-	-	{ }	true
struct *p = malloc	AddrStmt	$Expr("p")$ p=getMemObjAddress(malloc)	{ $p \rightarrow$ p, p$\rightarrow o_1$ }	true
int a	AddrStmt	$Expr("a")$	{$p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a }	true
int* x = &a	AddrStmt	$Expr("x")$ x=getMemObjAddress(&a)	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ }	true
*x = 5	StoreStmt	LocMap[x]=5	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ }	true
int* q = &(p−>f0)	StoreStmt	$Expr("q")$ q=LocMap[p+f0]	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ q$\rightarrow o_3$ }	true
*q = 10	StoreStmt	LocMap[q]=10	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ q$\rightarrow o_3$ }	true
int** r = &(p−>f1)	StoreStmt	$Expr("r")$ r=LocMap[p+f1]	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ q$\rightarrow o_3$, r$\rightarrow o_4$ }	true
*r = x	StoreStmt	LocMap[r]=x	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ q$\rightarrow o_3$, r$\rightarrow o_4$ }	true
int* y = *r	LoadStmt	Expr("y") y=LocMap[r]	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ q$\rightarrow o_3$, r$\rightarrow o_4$ $y \rightarrow$ y,y$\rightarrow o_4$}	true
int z = *q + *y	BinaryOPStmt	z==LocMap[q]+LocMap[y]	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ q$\rightarrow o_3$, r$\rightarrow o_4$ $y \rightarrow$ y,y$\rightarrow o_4$, z$\rightarrow$ 15}	true
@assert(z == 15)	Call for Solve	res==ite(z==15,true,false)	{ $p \rightarrow$ p, p$\rightarrow o_1$ $a \rightarrow$a,$x \rightarrow o_2$ q$\rightarrow o_3$, r$\rightarrow o_4$ $y \rightarrow$ y,y$\rightarrow o_4$, z$\rightarrow$ 15, res$\rightarrow$true}	SAT

Table 5.4: Example (4) SSE States Transitions

Figure 5.5 shows the example of function call and branch. To be noted, variable a was assigned with the value from $bar(1)$. The path condition is in $line4$, although the value expression has already stored in memory table, two paths should be translated and encoded. When the analyzer comes to the branch condition, the solver would add the path condition and check whether it is satisfied with the path condition. Firstly, we assume the condition is `true` and we send to solver. The analyzer would continue

```
1 int main(int argv) {
2     int a = bar(1);
3     int b = 0;
4     if(a > 3)
5         b = 10;
6     assert(b>=5);
7 }


10 int bar(int argv) {
11     return 5*argv;
12 }
```

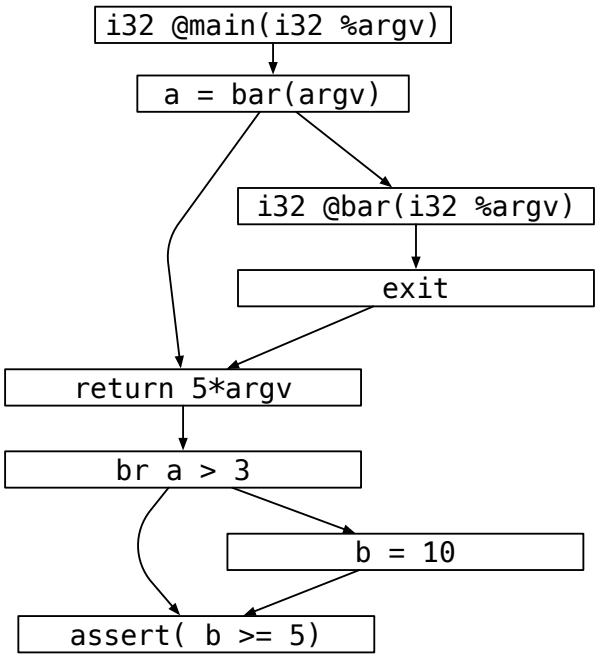

Figure 5.5: Example (5) Call

because the condition passes the solver. On the other hand, when the condition is `false`, the analyzer would not perform symbolic execution any more.

5.2 Optimization and Future Work

In this chapter, some interesting ideas are summarized, which will be implemented in our future work to improve the robustness and usefulness of previously demonstrated SSE. At the highest level of concept, SSE takes the fully single path based translation rules for symbolic execution states in memory table. This makes redundant costs for searching and traversal for multiple times in the beginning of the **ICFG**.

State forking: When an operation is access to an allocation expression, the state is forked in considering all possible states that may be resulted from the operation. And the path constraints can be explicitly explored and updated in the memory table for the expressions.

ICFGNode	SVFStmt	Operation	Constraints	Reasoning
i32@main()	-	-	{ }	true
int a = bar(1)	AddrStmt	$Expr("a")$	{a →a }	true
int bar(int argv)	CallPE	$Expr("argv") == 1$	{a →a, $argv$ →argv argv→ 1}	true
return 5 ∗ argv	RetPE	a==5∗argv	{a →a, $argv$ →argv argv→ 1, a→ 5}	true
int b = 0	AddrStmt	$Expr("b") == 0$	{a →a, $argv$ →argv argv→ 1, a→ 5, b→ 0}	true
if(a > 3)	BranchStmt	ite(a>3,true,false) == true *Solver Check*	{a →a, $argv$ →argv argv→ 1, a→ 5, b→ 0, (a > 3) ==true}	true
(if-then) b = 10	StoreStmt	b == 10	{a →a, $argv$ →argv argv→ 1, a→ 5, b→ 10, (a > 3) == true }	true
@assert(b >= 5)	Call for Solve	res==ite(b>=5,true,false)	{a →a, $argv$ →argv argv→ 1, a→ 5, b→ 10, (a > 3) == true, res→ true}	SAT
(if-else) @assert(b >= 5)	Call for Solve	ite(a>3,true,false) == false res=ite(b>5,true,false)	{a →a, $argv$ →argv argv→ 1, a→ 5, b→ 0, (a > 3) == false, res→ $false$}	UNSAT

Table 5.5: Example (5) SSE States Transitions

If-then-else: SSE now is separately processing the path according to each BRANCH instruction, which leads to path explosion as the program path goes deeper. The number of paths is exponentially growing and couldn't be fully processed by **SSE** when given the certain limited time. An alternative approach is to encode the uncertainty of the possible values into $Expr$ in which an efficient way to keep them in LocMap, without traversing and translating again for each branch.

5.3 Conclusions

Static Symbolic execution is a static program analysis technique that the program is analyzed by using symbolic values in representing the program behaviors. Exploiting from the **ICFG** paths allows us to automatically check and verify the existed and non-existed program faults.

In this dissertation, I demonstrated the framework and implementation of static symbolic execution. I also formulate a simplified LLVM-like language to clearly and precisely navigate our analysis targets. Our symbolic execution tool accepts LLVM-IR codes which is broadly used in various compilation tasks. This makes possible in supporting cross-language analysis.

We detailed the construction of the **ICFG** in performing context sensitive analysis. Our generic SSE can solve the program inter-procedural control flow reachability problem and assist to interpret the program in program properties. The validation of our tool was given.

Future development should focus in solving the exhaustive and repeatedly searching strategies. The current SSE doesn't apply any search strategies to reuse the previously explored edges during the collection of the paths phase. And on the other hand, the current memory table is straightforward and naive to annotate all program variables to their expressions, while most of them actually can be assumed as "dead" variables because they are never used or accessed. So that the next generation of the symbolic execution should make an effective method to incorporate with a smaller sized and "lively" memory table to be used during the analyzing.

Bibliography

[1] A. V. AHO, M. S. LAM, R. SETHI, AND J. D. ULLMAN, *Compilers: Principles, Techniques, & Tools*, Pearson Education India, 2007.

[2] F. E. ALLEN, *Control flow analysis*, ACM Sigplan Notices, 5 (1970), pp. 1–19.

[3] F. E. ALLEN AND J. COCKE, *A program data flow analysis procedure*, Communications of the ACM, 19 (1976), p. 137.

[4] J. R. ALLEN, K. KENNEDY, C. PORTERFIELD, AND J. WARREN, *Conversion of control dependence to data dependence*, in Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1983, pp. 177–189.

[5] G. BALATSOURAS AND Y. SMARAGDAKIS, *Structure-sensitive points-to analysis for C and C++* , in SAS '16, Springer, 2016, pp. 84–104.

[6] R. BALDONI, E. COPPA, D. C. D'ELIA, C. DEMETRESCU, AND I. FINOCCHI, *A survey of symbolic execution techniques*, ACM Comput. Surv., 51 (2018).

[7] R. BALDONI, E. COPPA, D. C. D,ÄÔELIA, C. DEMETRESCU, AND I. FINOCCHI, *A survey of symbolic execution techniques*, ACM Comput. Surv., 51 (2018).

[8] T. BEN-NUN, A. S. JAKOBOVITS, AND T. HOEFLER, *Neural code comprehension: a learnable representation of code semantics*, in NeurIPS '18, 2018, pp. 3585–3597.

[9] C. CADAR, D. DUNBAR, AND D. ENGLER, *Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs*, in Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI,Äô08, USA, 2008, USENIX Association, p. 209,Äì224.

[10] C. CADAR, V. GANESH, P. M. PAWLOWSKI, D. L. DILL, AND D. R. ENGLER, *Exe: Automatically generating inputs of death*, ACM Trans. Inf. Syst. Secur., 12 (2008).

[11] C. CADAR AND K. SEN, *Symbolic execution for software testing: Three decades later*, Commun. ACM, 56 (2013), p. 82,Äì90.

[12] L. A. CLARKE, *A system to generate test data and symbolically execute programs*, IEEE Trans. Softw. Eng., 2 (1976), p. 215,Äì222.

[13] P. GODEFROID, N. KLARLUND, AND K. SEN, *Dart: directed automated random testing*, in Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp. 213–223.

[14] S. GULWANI, S. SRIVASTAVA, AND R. VENKATESAN, *Program analysis as constraint solving*, in Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008, pp. 281–292.

[15] T. HANSEN, P. SCHACHTE, AND H. SØNDERGAARD, *State joining and splitting for the symbolic execution of binaries*, in International Workshop on Runtime Verification, Springer, 2009, pp. 76–92.

[16] D. HAREL, D. KOZEN, AND J. TIURYN, *Dynamic logic*, in Handbook of philosophical logic, Springer, 2001, pp. 99–217.

[17] J. C. KING, *Symbolic execution and program testing*, Commun. ACM, 19 (1976), p. 385,Äì394.

[18] C. LATTNER, *LLVM and Clang: Next Generation Compiler Technology LLVM: Low Level Virtual Machine*, tech. rep., 2008.

[19] Y. LEI AND Y. SUI, *Fast and precise handling of positive weight cycles for field-sensitive pointer analysis*, in SAS '19, Springer, 2019, pp. 27–47.

[20] L. LI, C. CIFUENTES, AND N. KEYNES, *Boosting the performance of flow-sensitive points-to analysis using value flow*, in FSE '11, 2011, pp. 343–353.

[21] J. LIBBY, K. KENT, ET AL., *A survey of data dependence analysis techniques for automated parallelization*, (2007).

[22] R. MARTIN, S. BARNUM, AND S. CHRISTEY, *Being explicit about security weaknesses*, CrossTalk The Journal of Defense Software Engineering, 20 (2007), pp. 4–8.

[23] T. REPS, *Program analysis via graph reachability*, in Proceedings of the 1997 International Symposium on Logic Programming, ILPS '97, Cambridge, MA, USA, 1997, MIT Press, p. 5,Äì19.

[24] O. G. SHIVERS, *Control-flow analysis of higher-order languages or taming lambda*, Carnegie Mellon University, 1991.

[25] Y. SUI AND J. XUE, *Svf: Interprocedural static value-flow analysis in llvm*, in Proceedings of the 25th International Conference on Compiler Construction, CC 2016, New York, NY, USA, 2016, Association for Computing Machinery, p. 265,Äì266.

[26] ——, *Svf: interprocedural static value-flow analysis in llvm*, in Proceedings of the 25th international conference on compiler construction, ACM, 2016, pp. 265–266.

[27] Y. SUI, D. YE, AND J. XUE, *Detecting memory leaks statically with full-sparse value-flow analysis*, IEEE Transactions on Software Engineering, 40 (2014), pp. 107–122.

[28] A. TARSKI, *A lattice-theoretical fixpoint theorem and its applications.*, Pacific journal of Mathematics, 5 (1955), pp. 285–309.

[29] S. YANG, D. YAN, H. WU, Y. WANG, AND A. ROUNTEV, *Static control-flow analysis of user-driven callbacks in android applications*, in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, IEEE, 2015, pp. 89–99.