University of California
Santa Barbara

Symbolic Quantitative Analysis for Software Testing and Security

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Seemanta Saha

Committee in charge:

Professor Tevfik Bultan, Chair
Professor Chandra Krintz
Professor Christopher Kruegel
Professor Ben Hardekopf

September 2022

The Dissertation of Seemanta Saha is approved.

Professor Chandra Krintz

Professor Christopher Kruegel

Professor Ben Hardekopf

Professor Tevfik Bultan, Committee Chair

September 2022

Symbolic Quantitative Analysis for Software Testing and Security

In memory of my beloved uncles

Nani Gopal Saha

and

Pankaj Saha

Acknowledgements

I would like to express my sincere thanks to my Ph.D. advisor, Tevfik Bultan, whose support, encouragement and mentorship have been invaluable to me. Throughout the years in my Ph.D. program and in the Verification Lab at UCSB, Tevfik has been exceptionally motivating, supportive, generous and a great mentor with his guidance and directions. It has been a great joy to learn about academic research, practical applications of research, leadership, mentoring and life skills from him.

I am thankful to my committee members, Chandra Krintz, Ben Hardekpof and Christopher Krugel, for providing me with feedback, support and encouragement to reach the milestones in my Ph.D.

My time as a graduate student researcher at UCSB was deeply enriched by the amazing group of researchers I collaborated with. I would especially like to thank Lucas bang, to whom I am very much grateful to have had the opportunity to work with and learn about solving research problems and writing papers. I would also like to thank William, Burak and Tegan for their collaboration, communication and great insights. The perspectives, company and support of the members of the Verification Lab: Nico, Nestan, Lucas, Tegan, William, Burak and Issac have made my time in lab intellectually exciting. I am also very thankful to Mara, Laboni, Shafiuzzaman, Shihua, Surendra, Ganesh, Albert, Chaofan and the ERSP students to make me more thoughtful as a researcher.

I am deeply grateful to Ripon Saha for his mentrorship, encouragement and advice towards me throughout the Ph.D. journey. I am also thankful to my internship manager, Mukul Prasad, from whom I have learned a lot of research perspectives.

I would like to thank all my friends and family from the Bengali community in Santa Barbara whose tremendous support has made my journey very smooth here in US. Some of these people have treated me in a way that I could hardly miss my family being in US. Their guidance is invaluable to me and I don't have enough words to thank them all.

I would like to thank my wonderfully supportive family, especially my exceptionally amazing parents, inspiring siblings and cousins, uncles, aunts, nephews, nieces and my extended new family. Your belief, kindness and true inspiration is everything to me.

Last but not the least, I would like to express my deepest thanks to my lovely wife who has supported me during the hard time of my last year of Ph.D. and prayed for my upmost success.

Curriculum Vitæ
Seemanta Saha

Education

2022	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2021	M.Sc. in Computer Science, University of California, Santa Barbara.
2012	B.Sc. in Computer Science and Engineering, Khulna University of Engineering & Technology.

Publications

Seemanta Saha, Laboni Sarker, Md Shafiuzzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, Tevfik Bultan. *Rare Path-guided Fuzzing.* (*under submission*)

Seemanta Saha, Surendra Ghentilya, Shihua Lu, Lucas Bang, Tevfik Bultan. *Obtaining Information Leakage Bounds via Approximate Model Counting.* (*under submission*)

Seemanta Saha, Mara Downing, Tegan Brennan, Tevfik Bultan. *PReach: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements.* Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, Pennsylvania, USA (**ICSE 2022**).

Tegan Brennan, **Seemanta Saha**, Tevfik Bultan. *JVM Fuzzing for JIT-Induced Side-Channel Detection.* In Proceedings of the 42nd International Conference on Software Engineering, Seoul, South Korea (**ICSE 2020**).

William Eiers, **Seemanta Saha**, Tegan Brennan, Tevfik Bultan. *Subformula Caching for Model Counting and Quantitative Program Analysis.* Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, California, USA (**ASE 2019**).

Seemanta Saha, Ripon K. Saha, Mukul R. Prasad. *Harnessing evolution for multi-hunk program repair.* Proceedings of the 41st International Conference on Software Engineering, Montreal, Quebec, Canada (**ICSE 2019**).

Seemanta Saha, William Eiers, Ismet Burak Kadron, Lucas Bang, Tevfik Bultan. *Incremental Attack Synthesis.* ACM SIGSOFT Software Engineering Notes 44 (4), 16, Proceedings of the 2019 JavaPathfinder Workshop, San Diego, California, USA, (**JPF 2019**).

Seemanta Saha, Ismet Burak Kadron, William Eiers, Lucas Bang, Tevfik Bultan. *Attack Synthesis for Strings using Meta-Heuristics.* ACM SIGSOFT Software Engineering Notes 43 (4), 56, Proceedings of the 2018 JavaPathfinder Workshop, Orlando, Florida, USA (**JPF 2018**).

Tegan Brennan, **Seemanta Saha**, Tevfik Bultan, Corina S. Pasareanu. *Symbolic Path Cost Analysis for Side Channel Detection.* Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands (**ISSTA 2018**) 27–37.

Tegan Brennan, **Seemanta Saha**, Tevfik Bultan. *Symbolic Path Cost Analysis for Side-Channel Detection.* Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, Gothenburg, Sweden (**ICSE 2018 Companion Volume Poster Track**) 424–425.

Abstract

Symbolic Quantitative Analysis for Software Testing and Security

by

Seemanta Saha

Quantitative program analysis is an emerging area with applications to software testing and security. In recent years, symbolic quantitative program analysis techniques based on symbolic execution and model counting constraint solvers have been applied to reliability analysis, performance evaluation, information flow analysis, side-channel detection and attack synthesis.

In this thesis, I focus on two significant problems in software testing and security: 1) assessment and guidance for testing techniques, and 2) assessment of information leakage. First, I present symbolic quantitative analysis techniques for assessment and guidance of testing techniques by identifying hard-to-reach statements and rare paths in programs, and guiding testing techniques using these information. Then, I present symbolic quantitative analysis techniques for quantifying the amount of information a program can leak, and synthesizing attacks to quantify the maximum amount of information that can be obtained by an attacker.

Towards assessing testing difficulty, I develop an efficient and scalable heuristic for probabilistic reachability analysis using branch model counting, dependency analysis, abstract interpretation and probabilistic model checking. The technique I develop can identify hard-to-reach program statements with high precision and accuracy compared to existing techniques based on symbolic execution and statistical sampling.

I also develop heuristics to identify rare program paths (program paths that a testing technique is unlikely to explore by generating random inputs) using control flow analysis, dependency analysis and branch model counting. Guiding concolic execution using the rare paths in the program, I can generate inputs that existing coverage-guided mutation-based fuzzing tools cannot.

Providing these inputs as the initial seed set to existing fuzzers increases code coverage.

Towards quantifying information leakage, I extend the existing symbolic quantitative analysis techniques by providing a framework to compute information leakage bound in presence of approximate model counting. I also generalize the existing side-channel attack synthesis techniques by providing support for unbounded strings using meta-heuristic search and improve efficiency of attack synthesis by exploiting the incremental nature of attack synthesis techniques.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Modern world is heavily dependent on software running on an increasingly large number of computing systems surrounding us. As the amount of software in safety critical systems such as cars, planes and medical equipments keeps increasing, assuring software quality has become one of the most fundamental problems that we are facing in this computing dominated era. Moreover, cyber-attacks stealing secret information from these systems are becoming immensely dreadful to the society. One can read about dependability and security problems caused by software vulnerabilities in the news everyday. It is extremely crucial to develop techniques that can improve the quality of software systems both in terms of dependability and security. In this thesis, towards advancing the development of dependable software systems, I focus on assessing difficulty of software testing techniques and guiding these techniques for better performance. Towards the development of secure systems, I advance the techniques for quantifying information leakage and synthesizing attacks in software.

1.1 Challenges in software testing

The most popular and effective approach to achieve dependability in software systems is software testing. Abundance of software testing techniques have been developed over the last 50 years. Although there has been a surge of progress in automated software testing in the last two decades based on random testing, fuzzing and symbolic execution, there are remaining challenges.

Automated testing techniques. Automated testing techniques can be classically separated as either concrete or symbolic. On one side, concrete testing techniques like fuzzing [1,2] and random testing [3] are very much scalable, can handle complex data structures but have difficulties in exploring hard-to-reach program statements and exercising paths guarded by magic numbers and complex checks [4–6]. On the other side, symbolic execution based techniques [7–9] can explore program paths guarded by complex checks by collecting and solving path constraints. However, these techniques are not as scalable as concrete testing techniques due to the exponential increase in the number of program paths in presence of loops and recursions. These techniques also suffer from high computational complexity due to constraint solving [10].

Hybrid testing techniques [6, 11–14] combine concrete (e.g., random testing [3], fuzzing [1]) and symbolic techniques (e.g., symbolic execution [8, 9], concolic execution [15, 16]) in order to improve testing performance and effectiveness. Typically, a strategy function for hybrid testing decides when to apply concrete techniques and when to apply symbolic techniques to achieve scalable and effective exploration of the program statements.

Assessment of testing difficulty. In order to choose between concrete and symbolic approaches, most existing strategies assess the difficulty of concrete testing based on the saturation of random testing [11, 13] or probabilistic program analysis [6, 14]. Assessment of testing difficulty is a significant problem. An efficient assessment of testing difficulty can provide useful guidance for combining different testing techniques and enable development of more efficient testing techniques.

Existing techniques [17, 18] based on symbolic execution and statistical sampling can be used to identify the program statements that are hard to reach. However, these techniques have scalability and complexity issues. I discuss these techniques below.

Probabilistic symbolic execution. Probabilistic symbolic execution [19] is an extension of symbolic execution that computes probabilities of program paths. It first uses symbolic execution [7] to collect path constraints and then use model counting constraints solvers [20] to compute path probability. Probabilistic symbolic execution is used for probabilistic reachability analysis, reliability analysis of software and software performance evaluation. However, probabilistic symbolic execution suffers from the same limitations as symbolic execution: it can only analyze program behaviors up to a certain fixed execution depth and the cost of symbolic execution increases exponentially with increasing execution depth due to exponential increase in the number of paths.

Statistical symbolic execution. Statistical symbolic execution [21] is more efficient and scalable compared to probabilistic symbolic execution [19]. However, it cannot compute precise probabilities rather provides approximate reachability probabilities with statistical guarantee. It samples over symbolic paths and provides statistical guarantee for probabilistic analysis. Statistical symbolic execution suffers from similar issues as probabilistic symbolic execution. There are two variants of statistical symbolic execution: 1) statistical analysis based on Monte Carlo sampling of symbolic paths, and 2) hybrid analysis combining both statistical and exact analysis based on informed sampling. One of the drawbacks of Monte Carlo sampling is that it needs to sample a large number of paths to achieve high statistical confidence. On the other side, informed sampling obtains more precise results and converges faster than Monte Carlo-based statistical analysis, however, its effectiveness suffers when the number of program paths grows exponentially.

Heuristic for Probabilistic Reachability. In this thesis, I develop a heuristic for probabilistic reachability analysis [22]. The technique can determine the likelihood (or, conversely, difficulty) of reaching a program statement if the program is being tested on randomly generated inputs. Using the probabilistic reachability analysis, I can identify hard-to-reach program statements for random testing techniques such as a fuzzer generating random inputs [1, 2]. From the experimental evaluation on software verification competition benchmarks [23], Apache Commons Lang [24] and Darpa benchmarks [25], I find that my technique achieves better precision and accuracy compared to both probabilistic symbolic execution and statistical symbolic execution and at the same time, more efficient. In particular, my technique 1) can model behaviors of arbitrarily long paths, 2) does not suffer from path explosion, i.e., the cost of my analysis increases in polynomial time with the size of the program (and does not depend on the execution depth) [26], and finally, 3) solves branch constraints rather than path constraints, reducing the cost of constraint solving. I will discuss this technique in detail in Chapter 3.

Guidance for testing techniques. The next step to improve automated testing techniques is to guide these techniques based on their difficulty assessment results. For example, if we have a technique to identify hard to reach program statements for a fuzzing technique, how can we use this information to guide the fuzzing technique for better performance?

Existing fuzzing techniques [4, 6] identify rare branches in the program and then either use mutation strategy [4] or symbolic execution [5, 6] to generate inputs to pass through the rare branches. These techniques use execution samples collected during fuzzing to identify rare program branches [4, 6]. As a result, these techniques suffer from the drawbacks of sampling based techniques, requiring large number of samples to provide statistically guarantee. Moreover, both mutation strategies and symbolic execution has their own drawbacks. Mutation based techniques [4] require extra instrumentation and analysis to generate input to pass through rare branches and symbolic execution based techniques [5, 6] suffer from path explosion problem.

Rare path-guided fuzzing. In this thesis, I develop a heuristic to identify rare program paths to guide mutation-based fuzzers. The technique guides the fuzzer to explore paths guarded by complex checks and magic numbers. I also develop algorithms to guide concolic execution using these rare paths to generate inputs that can execute the identified rare paths in the program. These inputs are provided as the initial seed set to the existing mutation-based fuzzing techniques [1, 4]. Experimental results on a benchmark consisting programs which are heavily dependent on structured inputs show that the generated inputs improve coverage performance of the fuzzers compared to a random initial seed. I will discuss about rare path-guided fuzzing in Chapter 4.

1.2 Challenges in quantifying information leakage

Trusting software systems with our sensitive and confidential information such as financial information, personal communication, physical location, or medical records has become very common these days. One of the most critical security issues in software systems today is to protect users confidential and sensitive information. Hence, it is crucial for software developers and engineers to write code in a manner that prevents leakage of sensitive data to unauthorized users.

Quantifying information leakage. A classic approach to address the problem of information leakage is enforcing noninterference which ensures that public output values are independent of secret input values. However, enforcing noninterference is often not possible and practical as software systems need to reveal some amount of information that depends on secret inputs. Consider a password checker where, as public output, the system provides information whether the user input matched or did not match the secret password. The public output of a password checker is dependent on the password (secret input), hence any password checker violates noninterference. Similarly, consider an online voting application which records secret ballots of individuals. The public result of the voting will depend on the votes of individuals which means that noninterference cannot be enforced. However, the privacy of each individual voter should be protected.

Noninterference, like many other classic program analysis problems, requires a binary answer which is not suitable for many practical scenarios as I discussed above. Recently, there has been increasing work on quantitative program analysis techniques where the goal is not only to give a binary "yes" or "no" answer, but also to quantify the result. For example, instead of using the binary noninterference property, which asks *is there any information leakage?*, I can ask a quantitative question: *how much information is leaked*?

Numerous techniques have been developed in recent years to quantify information leakage in software either by measuring the number of tainted bits, channel capacity or Shannon entropy [27–30, 30–44]. A large number of works [30, 33–37] use information theoretic approaches for quantifying information leakage. Most of these approaches use Shannon entropy-based information leakage computation [38–44] based on enumeration [38] or model counting [39–47] to compute information leakage. In this thesis, I also take the route of Shannon entropy-based information leakage computation. I use symbolic execution to collect path constraints and model counting to compute path probability.

Existing techniques to quantify information leakage using symbolic execution and model counting are limited to programs and constraints generated from the programs that can be supported using exact model counting tools [45, 46]. To support a wide range of programs and generated constraints, these techniques need to be extended by integrating other existing model counting techniques, for example, approximate model counting techniques supporting bitvector and floating point constraints [48, 49]. However, approximate model counting techniques do not provide the exact model count rather a lower and an upper bound of the exact model count. Moreover, symbolic execution often suffers from path explosion leading to partial path coverage.

The critical question to answer here is: how to compute bounds for information leakage given bounds for model count and partial information about the program paths.

Bounding information leakage. In this thesis, I develop symbolic quantitative program analysis techniques to quantify information leakage in presence of these two key sources of unsoundness: 1) partial path coverage by symbolic execution and 2) approximate model counts. I approach the problem of quantifying information leakage in presence of partial path coverage and approximate model counts as optimization problems. Then, I use existing optimization techniques to compute information leakage bounds. I will discuss the technique to bound information leakage in Chapter 5.

Side-channel attack synthesis Software side-channel attacks [50–52] are a class of vulnerabilities that leak information about secret values used in the software program. If a program is vulnerable to side-channel, an unauthorized user can capture secret information by observing non-functional properties of software systems such as execution time, memory space used etc. Given a program that performs computation over secret values, unknown to an unauthorized user, he or she can synthesize attacks to leak information about the secret values. The attacks often consist of a sequence of public inputs that a malicious user can feed to the program to leak information about secret values. Existing techniques to synthesize software side-channel attack are limited to either programs with integer values and generating bit-vector constraints [53] or string values for specific type of side channel (e.g. segment oracle attack) [39, 41].

Synthesizing attacks for program using metaheuristics. In this thesis, I develop a general approach for synthesizing attack using meta-heuristic search techniques handling constraints on integers, unbounded strings and mixed values [43]. The technique I develop can automatically synthesize side-channel attacks against string manipulating programs. I also develop an incremental approach of attack synthesis [44] that exploits the incremental nature of constraint solving during the meta-heuristic search phase. By synthesizing an attack, I provide a demonstration of side-channel vulnerability of a program. I will discuss these techniques in detail in Chapter 6.

1.3 Summary of Contributions and Thesis Outline

My thesis contributes the following:

- A heuristic for probabilistic reachability analysis to identify hard-to-reach program statements.
- A heuristic for rare path-guided fuzzing.
- Techniques for obtaining information leakage bounds via approximate model counting.
- Techniques for side-channel attack synthesis based on meta-hueristics and incremental analysis.

In the end, these contributions are aimed towards answering the considerations set forth in the prior section.

Towards advancing the development of dependable software systems my contributions are finding answers to the following questions: (1) is there a scalable and efficient approach towards assessment of testing difficulty, (2) can we guide testing techniques based on the assessment results.

Towards advancing the development of secure software systems my contributions are finding answers to the following questions: (1) how can we bound the amount of information leakage given approximation of model counts and partial path coverage, (2) how can we develop generalized and efficient attack synthesis techniques.

Rest of the thesis is organized as follows: I discuss about symbolic quantitative program analysis techniques and its applications in Chapter 2. I present heuristic for probabilistic reachability analysis in Chapter 3 and rare path analysis for fuzzing in Chapter 4. In Chapter 5 and 6, I present the techniques on bounding information leakage and side-channel attack synthesis respectively. I present all the related works in Chapter 7 and finally conclude my thesis in Chapter 8.

Chapter 2

Symbolic Quantitative Program Analysis

Quantitative program analysis is an emerging field with its applications to software reliability, quantitative information flow, software performance evaluation, side-channel detection and attack synthesis. Recently, symbolic quantitative program analysis techniques combining symbolic execution and model counting constraint solvers has been applied to software to check probabilistic properties, measure reliability of a software, performance analysis of software, quantify information leakage and synthesize attack for programs vulnerable to side-channel. The main concept behind symbolic quantitative program analysis is to generate constraints from programs by applying existing static analysis techniques such as symbolic execution, symbolic model checking, abstract interpretation etc. Once constraints are generated from the program, the next step is to use a model counting constraints solver to count number of satisfying solutions to the constraints. Then the counts is used to do probabilistic analysis such as path probability computation, quantifying bits of information etc. In this chapter, I will discuss probabilistic symbolic execution as a symbolic quantitative program analysis technique, statistical symbolic execution as an extended version of probabilistic symbolic execution and some of the

```
int checkPIN(int h[], int l[]) {
  for (int i = 0; i < 4; i++)
    if (h[i] != l[i])
      return 0;
    return 1;
}
```

Figure 2.1: Example program for symbolic quantitative analysis

applications of symbolic quantitative program analysis in detail.

2.1 Symbolic Execution

Symbolic execution engines are used to symbolically execute a program and collect the path conditions. Symbolic execution represents program's execution states symbolically. An execution state consists of a path condition and a mapping between program variables and their symbolic expressions over free variables and constants [19]. To reach a program point, certain branch conditions need to evaluate to true. A path condition is a conjunctive formula encoding all these branch conditions. Let us assume that to reach a program location, an input needs to pass through 3 branch conditions b_1, b_2 and b_3 where b_1, b_2 and negation of b_3 need to be true. Then, the path condition to reach that program point would be $b_1 \wedge b_2 \wedge \neg b_3$. Symbolically executing program P using a symbolic execution engine we can collect a set of path conditions ϕ where ϕ_i represents i^{th} path condition. Consider the PIN checking program in Figure 2.1. Symbolically executing this program, I collect 5 execution path conditions as shown in Table 2.1.

In my thesis, I have worked with two most popular symbolic execution tools: Symbolic Pathfinder [9] and KLEE [8] for symbolically analyzing Java and C program respectively. I use Symbolic Pathfinder for quantifying information leakage in java programs and developing attack synthesis techniques. I also use this symbolic execution tool to compare to the heuristic for probabilistic reachability analysis, I develop, for the purpose of identifying hard-to-reach program statements. I use KLEE and extend it for quantitative program analysis and quantifying information leakage for C programs.

2.2 Model Counting constraint Solver

A model counting constraint solver computes the number of solutions for a given constraint within a given bound [46, 54–58]. Recently, model counting constraint solvers have been applied to automating quantitative software verification, analysis and security tasks. The goal in quantitative program analysis is not to just give a "yes" or "no" answer, but to also quantify the result. This type of analysis is crucial for many domains since "yes" or "no" answers may not be possible. Moreover, most symbolic execution tools cannot guarantee absence of an assertion failure in general since they search the state space up to a certain execution depth. When combined with a model counting constraint solver, a symbolic execution tool can quantify the likelihood of reaching an unexplored part of the state space, hence providing a probabilistic upper bound on observing an assertion violation. In this thesis, I use automata-based constraint solver and model counter (ABC) [46] and an approximate model counter SearchMC [49] to develop symbolic quantitative program analysis techniques.

Automata-Based Constraint Solving and Model Counting

Automata-Based Model Counter (ABC) is a constraint solver for string and numeric constraints with model counting capabilities [46]. The constraint language for ABC supports all numeric constraints solved by off the shelf constraints solvers as well as typical string operations such as *charAt, length, indexOf, substring, begins, concat,* $<$, $=$, etc. Given a constraint C, ABC constructs a multi-track deterministic finite automaton (DFA) A_C that characterizes all solutions for the constraint C, where $\mathcal{L}(A_C)$ corresponds to the set of solutions for C. For each string term γ or integer term β in the constraint grammar [59], ABC implements an automata constructor function which generates an automaton A that encodes the set of satisfying solutions for the term. Note that variables within string terms and integer terms appear in separate automata, as separate encodings are used for each (ASCII for strings, binary encoding for integers). ABC implements specialized DFA construction algorithms for atomic string operations. Boolean operations ($\wedge$, $\vee$, $\neg$) are handled using standard DFA operations (intersection, union, and complement, respectively).

ABC counts the number of models (solutions) for a constraint C by first constructing the corresponding automaton A_C and using the observation that number of strings of length k in $\mathcal{L}(A_C)$ is equal to the number of accepting paths of length k in the DFA A_C. Consequently, ABC treats the DFA A_C that results from solving C as a graph where DFA states are graph vertices and the weight of an edge is the number of symbols that have a transition between the source and destination vertices (states) of that edge. A dynamic programming algorithm that computes the kth power of the adjacency matrix of the graph is used to count the number of accepting paths in the DFA of length k (or less than or equal to k) [46].

In this thesis, I use ABC for developing techniques for difficulty assessment of testing, guiding testing techniques, quantifying information leakage and attack synthesis. I extend ABC for incremental solving of constraints [44] where ABC can store results from earlier model counting queries and reuse the results later on to increase efficiency of model counting for synthesizing attacks. I also contribute in the development of subformula caching for automata-based model counting [60] in ABC which increases the efficiency of model counting-based quantitative program analysis techniques.

SearchMC, an approximate model counter

SearchMC is an approximate model counter which is developed using XOR streamlining techniques. Input to SearchMC is either a SMT formula ot a CNF formula. It does not provide an exact model count rather a lower and an upper bound of the log base 2 of the exact model count. The bound of model count it provides comes with a given confidence level δ and an approximation tolerance ϵ. XOR streamlining by adding random parity constraints is an effective approach for approximation. But, XOR streamlining techniques for approximation require a prior hypothesis. SearchMC uses statistical estimation to continuously refine the probabilistic estimate of the model count for a constraint formula. At each step. SearchMC refines the estimate by adding k XOR constraints to the initial constraint formula and then enumerates solutions, up to maximum number of c solutions. This enumeration processes is defined as exhaust-up-to-c query. At a particular step, SearchMC chooses c and k based on the prior estimate. Then, it uses the query results to update the estimate (posterior estimate). SearchMC uses binomial distribution model for the estimation. If the XOR constraints are fully independent, the distribution model would be exact. But, as the model is not exact, there is no soundness guarantee, even probabilistically like other approximate model counter [48].

However, SearchMC provides a variant of the technique which provides probabilistic soundness guarantee at the expense of more queries to reach to a specific precision. If the true model count of a constraint formula, F is C_F then the guarantee for the approximate model count c is probabilistically sound as below:

$$Pr([(1+\epsilon)^{-1}C_F \leq c \leq (1+\epsilon)C_F]) > 0.6 \tag{2.1}$$

SearchMC operates on bit-vector constraints with support to floating points. In this thesis, I use SearchMC with KLEE for quantitative information flow analysis. The version of KLEE [61] I use for quantitative program analysis can generate integer constraints for certain classes of program conditions. If it can generate integer constraints, I use ABC for model counting but if it cannot generate integer constraints, I use SearchMC for model counting. As SearchMC provides bounds of the exact count, I develop techniques to compute bounds of information leakage given the bounds of model count.

2.3 Probabilistic Symbolic Execution

Probabilistic symbolic execution [19] is an extension of symbolic execution that computes probabilities of program paths. Probabilistic symbolic execution has been used to address several quantitative program analysis problems in recent years such as software reliability analysis [62], performance analysis [63], quantitative information flow analysis [35], and side-channel analysis [64]. Probabilistic symbolic execution combines both symbolic execution engine and model counting constraint solver to perform probabilistic analysis of software.

In probabilistic symbolic execution, after collecting the path conditions, model counting constraint solvers are used to compute probabilities of path conditions (Algorithm 4). A model counting constraint solver computes the number of inputs (called models) that satisfy a given path condition. The number of satisfying inputs, c_i for i^{th} path condition is then divided by the domain size of the inputs, D to compute execution path probability $p_i = \frac{c_i}{D}$.

To use a model counting constraint solver, path conditions are typically translated to SMT formulas. Using a symbolic execution tool it is possible to generate SMT formulas. Table 2.1 shows model counts for the collected path constraints and computed path probabilities with a domain size of 256 (considering both h and l can have binary values only) for the PIN checking example in Figure 2.1.

Algorithm 1 ProbabilisticSymbolicExecution(P)
Symbolically executes program P, collects set of path constraints, Φ and computes a set of path probabilities, p.

```
1: $\Phi \leftarrow \emptyset$
2: $\Phi \leftarrow$ SymbolicExecution($P$)
3: $D \leftarrow \Phi$
4: for $\phi_i \in \Phi$ do
5:     $c_i \leftarrow$ ModelCount($\phi_i$)
6:     $p_i = \frac{c_i}{D}$
7:     $p \leftarrow p \cup \{p_i\}$
8: return $p$
```

Table 2.1: Probabilistic symbolic execution results for the PIN checking example.

i	Path Constraint	Model Count	Probability
1	$l[0] \neq h[0]$	128	0.5000
2	$l[0] = h[0] \wedge l[1] \neq h[1]$	64	0.2500
3	$l[0] = h[0] \wedge l[1] = h[1] \wedge l[2] \neq h[2]$	32	0.1250
4	$l[0] = h[0] \wedge l[1] = h[1] \wedge l[2] = h[2] \wedge l[3] \neq h[3]$	16	0.0625
5	$l[0] = h[0] \wedge l[1] = h[1] \wedge l[2] = h[2] \wedge l[3] = h[3]$	16	0.0625

Probabilistic symbolic execution in [19] supports Java programs only using symbolic execution tool Symbolic Pathfinder [9] and model counting tool LattE [45]. Later, support for model counting string constraints [46] was added in [39]. In this thesis, I have implemented probabilistic symbolic execution support for C programs on top of symbolic execution tool KLEE [8] using a latest version of KLEE [61] and model counting constraint solvers ABC [46] and SearchMC [49]. The implementation supports both bit-vector and integer constraints generated from programs and hence probabilistic symbolic execution can be applied on a wide range of programs.

2.4 Statistical Symbolic Execution

Statistical symbolic execution [21] is a better scalable technique for probabilistic analysis of software. To address the scalability drawbacks of probabilistic symbolic execution, statistical symbolic execution performs Monte Carlo sampling over symbolic program paths and uses the information for hypothesis testing and Bayesian estimation for probabilistic reachability analysis. To speed up the convergence of the sampling process, a new sampling approach named informed sampling was introduced in statistical symbolic execution. Using informed sampling, state space of the program that can be explored by symbolic execution is pruned from statistical analysis and the execution for sampling is guided towards less likely paths. The informed sampling technique combines Bayesian estimation with partial exact analysis using symbolic execution. As a result, statistical analysis converges faster using the information from symbolic execution side.

Statistical symbolic execution is implemented on top of symbolic execution tool Symbolic Pathfinder [9]. In this thesis, I use statistical symbolic execution tool as a probabilistic reachability analysis technique to identify hard to reach program statements and compare to the heuristic that I develop for probabilistic reachability analysis.

2.5 Applications of Symbolic Quantitative Program Analysis

In this section, I will discuss some of the applications of symbolic quantitative program analysis in details.

Quantitative Assessment of Testing Difficulty

Software testing is the most popular approach for software quality assurance. Even after more than 50 years of active research in software testing, there are remaining challenges. Assessment of difficulty of testing techniques and guidance for improved performance is a significant problem to solve. Symbolic quantitative program analysis is an effective way to perform probabilistic analysis of software and provide quantitative results about different aspects of software such as probabilistic reachability analysis. In this thesis, I develop symbolic quantitative program analysis techniques for quantitative assessment of testing difficulty and then guiding testing techniques using the results from quantitative analysis.

In order to assess testing difficulty I develop a heuristic for probabilistic reachability analysis using a control flow analysis, model counting on branch constraints, abstract interpretation and probabilistic model checking. I use the heuristic to identify hard-to-reach program statements. Given a program and a target statement, the heuristic identifies if the target statement is easy to reach or hard to reach for a random testing tool or a fuzzing technique, generating random inputs. I will discuss this technique in detail in Chapter 3.

Guidance for Testing Techniques

Information from the quantitative assessment of testing difficulty can be used to guide testing techniques. For example, often in hybrid testing setup [5, 6, 11], random testing technique and symbolic execution are combined to use the best of both. On one side, random testing techniques are easy to run, work with complex data structures, scale for larger programs [3], often fail to explore program paths guarded by specific program conditions (e.g. the value of a variable should be a magic number). On the other side, symbolic execution explores path guarded by specific program conditions by collecting the path constraints and solving using a constraint solver [8, 9], but fails to analyze programs with exponential number of program paths and complexity of symbolic execution increases due to path constraint solving complexity [10]. If program paths that are hard to reach for a random testing tool can be identified beforehand and reduce the overload for symbolic execution to solve only the identified hard to reach paths instead of whole program, performance of both testing techniques can be maximized. In this thesis, I develop a symbolic quantitative analysis technique to identify hard to reach program paths for random testing technique such as coverage guided fuzzer [1] and then use symbolic execution to generate inputs to explore these hard to reach paths. And then, I provide these inputs as initial seed set to the fuzzer as guidance to explore hard-to-reach program paths as early as possible and later on mutate the inputs to explore comparatively easy to reach program paths. I will discuss this technique in detail in Chapter 4

Quantitative Information Flow Analysis

Quantitative information flow analysis of a program is done by extending probabilistic symbolic execution via observable tracking, model counting path constraints and information theoretic analysis using Shannon entropy. By tracking observable value for each path, it is possible to collect observation constraints. Number of observation constraints is always less than or equal to the number of path constraints for a program. Two or more path constraints with indistinguishable observable values are merged together in a disjunctive manner to construct one observation constraint. Observables can differ based on the information leakage problem in hand. For quantifying information leakage from the main-channel of a program (i.e. information leakage due to output values of the program), the observable can be simply the return value of the program. For detecting and quantifying information leakage due to side channels in a program, observable can be execution time for tracking timing side-channels, or the observable can be the size of memory allocated for space side-channels.

Two main building components of quantitative information flow analysis techniques are symbolic execution and model counting. Issues with existing techniques based on these two building components are: 1) symbolic execution can not analyze programs with exponential number of paths, hence, there can be unsoundness of information leakage quantified based on partial path coverage and 2) exact model counting tools cannot support constraints generated from a wide range of programs, hence, approximate model counters should be used. However, approximate model counters provide lower and upper bounds of true model counts. I will discuss the technique to deal with these two sources of unsoundness to quantify information leakage bounds for programs in chapter 5.

Attack Synthesis for Side-Channel Vulnerabilities

Attack synthesis techniques generate inputs in an iterative manner which, when fed to code that accesses the secret, reveal information about the secret based on the side-channel observations [40, 43, 44, 65]. Probabilistic symbolic execution is used to estimate probabilities of execution paths, and optimization techniques are used to maximize information gain based on entropy. Consider the pin checking function in Figure 2.1. The function has a timing side-channel and one can reveal the secret by measuring execution time. If h and l have no common prefix, the program will have the fastest execution since the loop body will be executed only once; If h and l have a common prefix of one index, a longer execution will be observed since the loop body executes twice. The case when h and l match completely, the program has the longest execution. An attacker can choose an input and use the timing observation to determine how much of a prefix of the input has matched the secret. Adaptive attack synthesis approach starts by automatically generating the path constraints using symbolic execution. It then uses these constraints to synthesize an attack which determines the value of the secret (h). Based on Shannon entropy, the remaining uncertainty of h can be computed to measure the progress of an attack.

Existing attack synthesis techniques can either deal with numeric program values or bounded string values. In this thesis, I develop a generalized technique using metaheuristic techniques, supporting unbounded string values. I will discuss my contribution to the development of side-channel attack synthesis techniques in more details in Chapter 6.

Reliability Analysis

One measure of program reliability is the probability that the program executes successfully. Probabilistic symbolic execution provides a means to compute program reliability. One run of probabilistic symbolic execution gives the probability that a randomly chosen input will execute that particular program path. By computing this probability for each complete program path, it is possible to determine what percentage of the input space is captured by the path constraints generated by probabilistic symbolic execution and therefore provide a lower bound on the reliability of the program.

As an example, consider the pin checking function in Figure 2.1. One can run bounded probabilistic symbolic execution and then compute what percentage of the input space leads to a program path that terminates within the depth bound. This gives the percentage of input space that can be confidently said will execute without failure and thus provides a lower bound on the reliability of the program. For the pin checking example, imagine we limit the search depth so that the loop symbolically executes only 3 times. In this case, all program paths for which the first three bits match would not complete their symbolic execution. Covered probability for reliability analysis will be then the summation of the probability of path constrains 1, 2 and 3 from Table 2.1.

In practice, we are also often interested in guaranteeing a lower bound for program reliability. In this case, we can perform model counting at each step of symbolic execution to determine what percentage of input follows which paths. This would allow us to guide the symbolic execution along the paths with higher probability in order to increase coverage most efficiently and stop execution once a certain coverage is reached. Conversely, one could also guide symbolic execution towards path with lower probabilities in order to test corner cases.

Performance Analysis

One of the significant software engineering tasks is to analyze performance and understand the distribution of program execution times for potential best-case and worst-case scenarios. Techniques based on traditional program analysis approaches collect program profiles to identify performance bottlenecks. But, this often fail to capture the overall performance of the program. Probabilistic symbolic execution is used to capture the probability distribution of inputs over execution times by exploring high- and low- probability paths. Once a path is explored, a set of test inputs can be generated and the program can be executed to model the performance of the program paths.

Chapter 3

Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements

Assessing difficulty of testing techniques is crucial to make the best use of the techniques. Although different testing strategies have been developed over the years and there has been immense progress towards software quality assurance, there are remaining challenges. For example, testing techniques generating inputs randomly are better scalable compared to other testing techniques and can handle complex data structures. But, these techniques can hardly explore program paths guarded by specific and complex program conditions.

In this chapter, I present a heuristic for probabilistic reachability analysis to identify *hard to reach* program statements towards assessing the difficulty of testing techniques generating inputs randomly [22]. There are existing techniques (e.g. probabilistic symbolic execution, statistical symbolic execution) that can be used for probabilistic reachability analysis to identify *hard to reach* program statements.

However, probabilistic symbolic execution suffers from the same limitations as symbolic execution: 1) It can only analyze program behaviors up to a certain fixed execution depth, hence it cannot analyze behaviors of arbitrarily large program paths. 2) Due to exponential increase in number of paths with increasing execution depth (path explosion problem), the cost of symbolic execution increases exponentially with increasing execution depth. 3) Although the sizes of path constraints generated by symbolic execution increase linearly with the execution depth, since the worst case complexity of constraint solvers is exponential, the linear increase in path constraint sizes can lead to exponential increase in analysis cost. Hence, path explosion combined with increasing sizes of path constraints can lead to double exponential blow up in the cost of symbolic execution, limiting its practical applicability.

Statistical symbolic execution suffers from similar issues as probabilistic symbolic execution. There are two variants of statistical symbolic execution: 1) statistical analysis based on Monte Carlo sampling of symbolic paths, and 2) hybrid analysis combining both statistical and exact analysis based on informed sampling. One of the drawbacks of pure statistical sampling is that it needs to sample a large number of paths to achieve high statistical confidence. Informed sampling obtains more precise results and converges faster than a purely statistical analysis, but its effectiveness suffers when the number of program paths grows exponentially.

The heuristic I develop addresses the above shortcomings of probabilistic symbolic execution and statistical symbolic execution. In particular, 1) my approach can model behaviors of arbitrarily long paths, 2) it does not suffer from path explosion, i.e., the cost of my analysis increases polynomially with the size of the program (and does not depend on the execution depth) [26], and finally, 3) it solves constraints arising from branch conditions rather than path constraints which reduces the cost of constraint solving.

My approach, which I implemented in a tool named PReach, works as follows. In order to compute reachability probability of statements, I introduce a concept called *branch selectivity* that determines the proportion of values satisfying a given branch condition. A branch is very selective if only a few values satisfy the branch condition. On the other hand, if a lot of values satisfy the branch condition, then the branch is not very selective. Given a target statement in a program, PReach identifies the input dependent branch conditions that influence the reachability probability of that statement using dependency analysis. Then, PReach constructs a discrete-time Markov chain model from the control flow graph of the program by computing branch selectivity of each branch condition that influences the reachability probability of the target statement. PReach uses abstract interpretation to determine the set of values that reach each branch condition and model counting to compute the branch selectivity value for each branch in the program that influences statement reachability. Finally, PReach uses a probabilistic model checker to compute the reachability probability of the target statement based on the constructed discrete-time Markov chain model.

One shortcoming of my approach is that it is not a sound program analysis technique and hence, it does not provide guarantees in terms of the precision or accuracy of the reachability probabilities it reports. On the other hand, probabilistic symbolic execution and statistical symbolic execution are not sound either since they can only analyze program behaviors up to a fixed execution depth.

I experimentally evaluate PReach on programs from the SV-COMP benchmark set used in Competition on Software Verification [23] and Competition on Software Testing [66]. Each program in this benchmark set contains an assert statement. I use these assert statements as the target of my probabilistic reachability analysis. I evaluate the effectiveness of my technique in separating *hard to reach* assert statements (i.e., assert statements with low reachability probability) from easy to reach assert statements (i.e., assert statements with high reachability probability) using a probability threshold (i.e., if the reachability probability of a statement is below the given threshold I classify it as *hard to reach*).

In order to determine the ground truth, I use a generator based random fuzzer that is based on JQF [67] and ZEST [68]. I set a time limit for the random fuzzer, and the assert statements that are not reached within the given timeout are marked as the *hard to reach* assert statements. Of the 142 programs I used in my experiments, the random fuzzer times out on 51 programs. PReach classifies the programs that the random fuzzer times out on as hard-to-reach, with

95.8% precision and 95.1% accuracy. In particular, my technique correctly classifies 135 out of 142 programs and generates only 2 false positives (reports *hard to reach* although the fuzzer does not time out) and 5 false negatives (reports *easy to reach* although the fuzzer times out).

In order to further evaluate the effectiveness of my probabilistic reachability analysis, I provide a detailed experimental comparison with the probabilistic symbolic execution (PSE) [19] and statistical symbolic execution (SSE) [21] extensions to Symbolic Pathfinder (SPF) [69] tool. Experimental results show that for programs with bounded execution depth, PSE achieves very high precision and accuracy to identify *hard to reach* cases. However, PReach outperforms PSE for programs with unbounded execution depth in terms of precision, accuracy and average analysis time. For large search depths PSE is unable to analyze 38% of the target programs demonstrating its limitations in terms of applicability and scalability, whereas PReach can analyze 100%. I compare PReach with SSE on the set of programs that PSE performs poorly. SSE was unable to analyze 27% of these programs and PReach outperforms SSE in terms of precision, accuracy, and average analysis time.

Finally, I analyze 24 target statements in 18 methods from Apache Commons Lang [24] and DARPA STAC Benchmarks [70]. PReach can classify 19 of the 24 target statements correctly demonstrating its effectiveness on real world programs, whereas PSE and SSE Ire able to successfully analyze and classify only one.

3.1 Overview and Motivation

I formalize probabilistic reachability analysis as follows. Given a program p, let i denote the input for the program, and I denote the domain of inputs (i.e., $i \in I$). Note that i can be a scalar value, a tuple, or a list of values. Given a target statement t in program p, the goal of probabilistic reachability analysis is to determine how likely it is to reach target statement t. I do this by determining how likely it would be to pick inputs that result in an execution that

```java
public class Main {
  public static void main(String[] args) {
    int arg = Verifier.nondetInt();
    if ( arg < 0 )
      return;
    int x = arg / 5;
    int y = arg / 5;
    Main inst = new Main();
    inst.test(x, y);
  }
  public void test(int x, int z) {
  System.out.println("Testing ExSymExe7");
    int y = 3;
    z = x - y - 4;
    if ( z != 0 )
      System.out.println("branch F001");
    else {
      System.out.println("branch F002");
      assert false;
    }
    if ( y != 0 )
      System.out.println("branch B001");
    else
      System.out.println("branch B002");
  }
}
```

Figure 3.1: An example based on SV-COMP benchmark

reaches t. In order to determine how likely it would be to pick such inputs, I determine the probability of picking such inputs if inputs are chosen randomly. I define $\mathcal{P}(p,t)$ as:

> $\mathcal{P}(p,t)$ denotes the probability of reaching statement s during the execution of program p on input i if i is selected randomly from the input domain I.

I assume uniform distribution of inputs in my current implementation. However, my technique can be easily extended to support any input distribution by integrating usage profiles [21] used in other probabilistic analysis techniques.

It is well-known that determining reachability of a statement in a program is an uncomputable problem. Hence, determining $\mathcal{P}(p,t)$ precisely is also an uncomputable problem. In this chapter I present a heuristic approach that approximates $\mathcal{P}(p,t)$. I report the reachability probability as a real number between 0 and 1.

Branch Selectivity My heuristic approximation of $\mathcal{P}(p, t)$ relies on a concept I call *branch selectivity.* Given a branch b, branch selectivity $\mathcal{S}(b)$ is proportional to the ratio of the number of values that satisfy the branch condition b to the total number of values in the domain of branch condition b. Formally, given a branch b, let D_b denote the Cartesian product of the domains of the variables that appear in b, and let $T_b \subseteq D_b$ denote the set of values for which branch b evaluates to true. Let $|D_b|$ and $|T_b|$ denote the number of elements in these sets, respectively. Then, $\mathcal{S}(b) = \frac{|T_b|}{|D_b|}$.

So, the selectivity of a branch gets closer to 0 as the number of values that satisfy the branch condition decreases, and it gets closer to 1 as the number of values that satisfy the branch condition increases. If we think of branch as a sieve, when $\mathcal{S}(b) = 0$ branch b does not allow any value to pass, and when $\mathcal{S}(b) = 1$ branch b allows all values to pass. Note that, if we pick values from the domain D randomly with a uniform distribution, then $|T_b|/|D_b|$ corresponds to the probability of picking a value that satisfies the branch condition. The branch becomes more selective as the probability of picking a value decreases.

An Example Consider the integer-manipulating program in Figure 3.1. This program is a modified version of an example from the *jpf-regression* directory of the SV-COMP benchmark used for software verification and testing competitions [23]. The target statement is the assertion statement in line 19. The *arg* variable's value is a randomly generated integer value and it denotes the input to this program. The question we want to answer for this program is: *how likely it is to reach the assertion statement at line 19 if we randomly generate values for the arg variable?*

The first conditional statement at line 4 ignores all the negative values. At line 15, possible values for z can be any randomly generated positive value, divided by 5, minus 7. Now, the assertion at line 19 is reachable when value of z is equal to 0. The likelihood of the value of z being equal to 0 is low if the input is a random number generated from a uniform distribution. Therefore, the probability of reaching the assert statement in this program is low.

My analysis uses branch selectivity based on model counting to successfully determine the reachability probability of the assert statement in this program. I inspect each branch condition leading to the assertion to determine how selective the branch is (i.e. what ratio of input values satisfy the branch). If I assume a domain of integer values, then for the conditional statement $arg < 0$, branch selectivity is calculated as half of the domain. Therefore, the possible values reaching the assertion is reduced to half. Next, for the next conditional statement, $z \neq 0$, branch selectivity is close to 1. Most values satisfy this constraint and conversely, only 1 value of z satisfies its negation. The assertion lies on the else branch of this condition, making it reachable only for one value of z.

Using the branch selectivity values computed at these branches, I convert the control flow graph of the program to a discrete time Markov chain as shown in Figure 3.3c. I use a probabilistic model checker to analyze the Markov chain and obtain a probabilistic measure for assertion reachability. For the running example, this reachability probability is computed as $0.5 \times (2.32e^{-10})$. The value 0.5 arises from the branch selectivity for the branch condition $arg < 0$ and $2.32e^{-10}$ arises from the branch selectivity for the branch condition $z \neq 0$. The reachability probability of the assertion statement is then reported as $1.16e^{-10}$, hence this statement would be classified as a *hard to reach* statement by my analysis since it has a low reachability probability.

To assess the success of my analysis for this example, I run a generator based random fuzzer with a timeout of 1 hour. I find that the fuzzer cannot generate an input to reach the assertion. The fuzzer generates 4,103,625 inputs and none of them reach the assertion, which supports the result of my analysis.

Since my analysis does not precisely represent the original semantics of the program, I cannot make soundness claims about the probability computed by my heuristic. In general case, my analysis may over or under approximate the reachability probability. By integrating abstract interpretation techniques to my analysis, I achieve better precision which I will discuss in section 3.2.

In the following sections I discuss how I compute and use branch selectivity values together with control flow, dependency analysis and abstract interpretation to extract a discrete-time Markov chain and then use probabilistic model checking to compute approximations of reachability probability.

3.2 A Probabilistic Reachability Heuristic

I approximate $\mathcal{P}(p,t)$ using a combination of control flow, dependency analysis, abstract interpretation, model-counting and probabilistic model checking. First, I discuss how model counting constraint solvers and abstract domains can be used to compute branch selectivity. Then, I use control flow and dependency analysis and branch selectivity to transform the program's control flow graph into a Markov chain. I form queries on this Markov chain solvable by probabilistic model checking whose solutions approximate $\mathcal{P}(p,t)$. If $\mathcal{P}(p,t)$ is less than a given threshold T_H, target statement is predicted as *hard to reach.* I discuss these steps below.

Branch Selectivity

The enabling technology for computing branch selectivity is model counting. Model counting is the problem of determining the number of satisfying solutions to a set of constraints. A model counting constraint solver is a tool which, given a constraint and a bound, returns the number of satisfying solutions to the constraint within the bound. For a branch condition b, recall that $\mathcal{S}(b) = \frac{|T_b|}{|D_b|}$, where D_b is the Cartesian product of the domains of the variables that appear in b and T_b is the set of values in D_b for which b evaluates to true. For a given b and D_b, a

```
public void test(int x) {
  if(x >= 0) {
    int y = -x;
    if (y > 0) {
      assert false;
    }
  }
}

```

(a) Using interval analysis

```
public void test(int x,
  int z, int r) {
  int y = 3;
  r = x + z;
  z = x - y - 4;
  if (x < z)
    assert false;
}

```

(b) Using relational analysis

Figure 3.2: Refined branch selectivity

model-counting constraint solver computes $|T_b|$. Then, using $|T_b|$ I compute $\mathcal{S}(b)$.

I use the Automata-Based Model Counter (ABC) tool, which is a constraint solver for string and numeric constraints with model counting capabilities [46]. The constraint language for ABC supports linear arithmetic constraints as well as typical string operations. In order to compute $\mathcal{S}(b)$ I first extract the branch condition from the program and then generate a formula in the SMT-LIB format that corresponds to the branch condition. Then, I send the formula to ABC as model counting query.

Refined Branch Selectivity

Abstract interpretation techniques overapproximate program behaviors by interpreting programs over abstract domains. key insight here is that it is possible to use abstract interpretation to refine and restrict the set of values that variables can take at each branch in order to better approximate the branch selectivity. Given a branch b, using abstract interpretation I generate a refinement condition R_b to overapproximate the set of values that the variables can take at that branch. R_b is then conjoined with T_b and D_b to compute refined branch selectivity $\mathcal{RS}(b)$. For a branch condition b, refined branch selectivity is defined as $\mathcal{RS}(b) = \frac{|T_b \wedge R_b|}{|D_b \wedge R_b|}$.

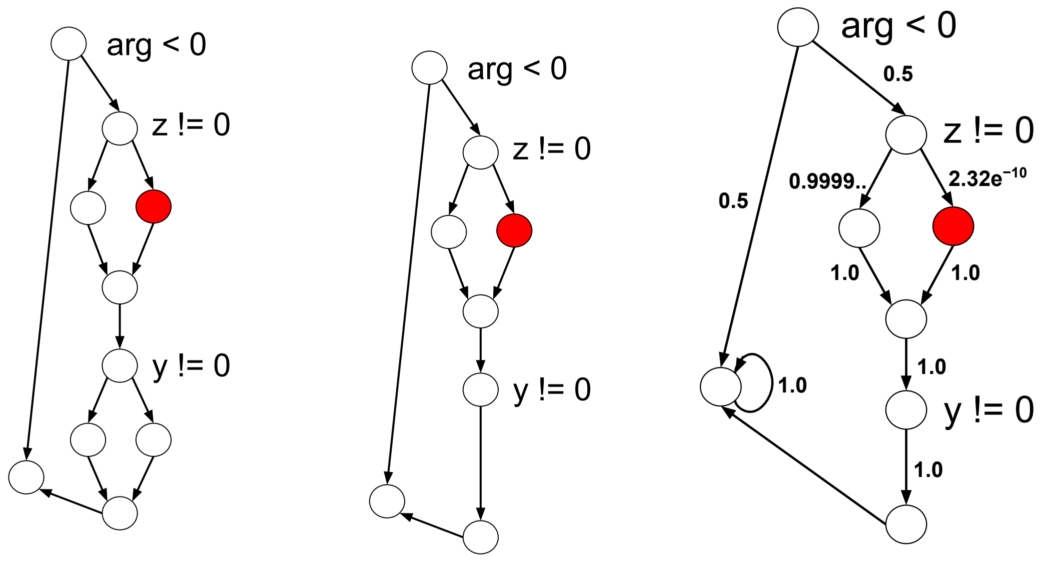
To implement the refined branch selectivity, I use state-of-the-art Java numeric analysis tool JANA [71] which supports two different abstract domains, intervals [72] and polyhedra [73], where polyhedra domain leads to more precise results however it is less scalable. I experimented with both of these domains to extract the refinement conditions R_b for each branch using interval analysis and relational (using polyhedra domain) analysis. I call these implementations PREACH-I and PREACH-P, respectively.

Consider the two code snippets from Fig. 3.2a and 3.2b. At line 4 in Fig. 3.2a, T_b and D_b are $y > 0$ and *True* respectively. $\mathcal{S}(b)$ computed by PREACH is 0.25 predicting incorrectly that the assertion is reachable. Applying either interval or relational analysis, R_b is extracted as $y < 0$ (at line 4, possible reachable values of x is greater than 0 and hence possible reachable values for y is less than 0 due to the update on variable y at line 3). T_b and D_b are updated as $y > 0 \wedge y < 0$ and $y < 0$ respectively using R_b and $\mathcal{RS}(b)$ computed by PREACH is 0 predicting correctly that the assertion is not reachable. Similarly, at line 6 in Fig. 3.2b, T_b and D_b are $x < z$ and *True* respectively. $\mathcal{S}(b)$ is computed as 0.5 predicting incorrectly that the assertion is reachable. Applying an interval analysis, there will be no refinement conditions as it is not possible to catch the relation between the variables x and z using the interval domain. But, applying relational analysis using the polyhedra domain, R_b is extracted as $x > z$ (possible reachable values of z is equal to $x - 7$). T_b and D_b are then updated as $x < z \wedge x > z$ and $x > z$ respectively and $\mathcal{RS}(b)$ is computed as 0, correctly predicting that the assertion is not reachable.

Target Statement Subgraph Extraction

The control flow graph of a program is a representation of all paths that may be traversed during execution. Given a program p, a target statement t in p and the input domain I, I extract the control flow graph of p, $\mathcal{G}(p)$, and mark the node of the control flow graph containing the target statement t as the node n^t.

I expedite my analysis by extracting the *target statement subgraph*, $\mathcal{G}(p,t)$ of $\mathcal{G}$. $\mathcal{G}(p,t)$



(a) Control flow graph

(b) Target statement subgraph

(c) Markov chain construction

Figure 3.3: Target statement subgraph extraction and Markov chain construction for the running example

contains all the control flow graph information needed to perform my analysis. I define this subgraph using standard concepts from control flow analysis. I define a *branch node* b in a control flow graph to be any node with more than one outgoing edge. The corresponding *merge node* m of a branch node b is its immediate post-dominator. The *component* C defined by b is the union of branch node b, its merge node m and all nodes of the control flow graph reachable from b without going through m. The *maximal component* of a node is the largest component containing that node. Any non-maximal component containing this node will be contained in this maximal component.

To extract $\mathcal{G}(p,t)$, I first find the maximal component of n^t. If n^t is not contained in any component, then n^t must lie on every path through $\mathcal{G}(p)$. Therefore, it is reached with certainty, $\mathcal{P}(p,t) = 1$, and my analysis can be terminated. Otherwise, the maximal component of n^t is the *maximal statement subgraph.*

$\mathcal{G}(p,t)$ is a subgraph of the maximal statement subgraph. To obtain $\mathcal{G}(p,t)$, I remove any component of the maximal statement subgraph that does not contain the statement node n^t. The branch and merge nodes of these components remain in the subgraph with one outgoing edge from the branch node to the merge node. $\mathcal{G}(p,t)$ results from this procedure.

Figure 3.3 shows the process of the target statement subgraph extraction on the running example from Figure 4.1. Figure 3.3a gives the control flow graph $\mathcal{G}(p)$ with the statement node n^t highlighted in red. Figure 3.3b shows the target statement subgraph $\mathcal{G}(p,t)$ extracted from $\mathcal{G}(p)$. In this example, the branch corresponding to $y \neq 0$ is removed from the control flow graph structure. The decision made at this branch does not impact the probability of reaching the target statement node.

Note that the target statement subgraph extraction phase is a heuristic to speed up my analysis. The subsequent stages can be performed on the entire control flow graph but this would result in unnecessary work including extra model counting queries which would slow down the analysis.

Markov Chain Construction

I define a weight for each edge of $\mathcal{G}(p,t)$. These weights transform $\mathcal{G}(p,t)$ into a Discrete Time Markov Chain (DTMC), $\mathcal{M}(p,t)$. A DTMC is a tuple $(S, \bar{s}, P, L)$ where S is a finite set of states, $\bar{s} \in S$ is the initial state, $P\colon S \times S \to [0,1]$ is the transition probability matrix where $\sum_{s' \in S} P(s,s') = 1$ for all $s \in S$. Each element $P(s,s')$ of the transition probability matrix gives the probability of making a transition from state s to state s'.

I use dependency analysis in the construction of the Markov Chain as I want to identify the branches dependent on input to set the weights of the edges accordingly.

Dependency Analysis A branch condition is input dependent if the evaluation of the condition depends on the value of the program input. Given a program and its marked input, I use static dependency analysis to identify the input dependent branches. Dependency analysis provides an over approximation of the set of branch conditions whose evaluation depends on the inputs. I use Janalyzer [74], an existing static analysis tool, to perform the dependency analysis. Janalyzer is implemented on top of the WALA [75] program analysis framework.

Then, I construct the Markov chain by assigning weights to each edge of $\mathcal{G}(p, t)$. $\mathcal{G}(p, t)$ is a directed graph: each edge begins at a source node s and ends at a destination node d. Given an edge $e : s \rightarrow d$: If e is the only edge beginning at s, the weight of e is 1. Else, s is a branch node by definition. To determine its weight I use a combination of dependency analysis and branch selectivity. Since b is a branch node, there is a branch condition associated.

- If the branch condition is independent from the program input, I weigh edge e as follows. Let E be the number of edges originating at s and $E^t \leq E$ be the number of edges originating at s which lie on a path to the target statement node n^t. If $E^t = 0$, then the weight of e is $1/E$. Otherwise, if e lies on a path to n^t weight of e is $1/E^t$. If e does not lie on a path to n^t, weight of e is 0.

- If the branch condition is dependent on the program input, I compute the weight of the edge e as follows. I use a model-counting constraint solver to determine the branch selectivity of b, $\mathcal{S}(b)$. If e is the edge corresponding to the if condition, the weight of e is $\mathcal{S}(b)$. Else, $1 - \mathcal{S}(b)$.

At the end of this phase, $\mathcal{G}(p, t)$ has been transformed into Markov chain $\mathcal{M}(p, t)$ where the probability of transitioning from one state to the next is given by the edge weight.

Figure 3.3c shows $\mathcal{M}(p, t)$ for the running example. The transition probabilities are given as edge weights. The two branch conditions yield the only non 1 edge weights in the graph. Both of these branch conditions are input dependent as determined by the dependency analysis. For each branch condition, the model-counting constraint solver ABC was used to find its branch selectivity. This selectivity was used to compute the weight of the edge corresponding to the if branch and its complement was used to compute the weight of the edge corresponding to the else branch.

PCTL Query Formulation

I automatically synthesize queries over $\mathcal{M}(p, t)$, whose solutions yield an approximation of $\mathcal{P}(p, t)$. The query I synthesize is:

- What is the probability that the target node n^t is reached at least once?

The answer to this query approximates $\mathcal{P}(p, t)$. I use a probabilistic model checker PRISM [76], a tool that analyzes systems that exhibit probabilistic behavior, to answer this query. I generate a discrete time Markov chain (DTMC) model based on the syntax supported by the PRISM tool. I can synthesize queries like *what is the probability of reaching a state in the Markov chain eventually?*.

In PRISM, a PCTL formula is interpreted over the DTMC model. Two types of formulas are supported: state formulas and path formulas where path formulas occur only when there is a probabilistic measure that needs to be included in the specification. For my analysis, the queries I synthesize are path formulas and are of the form $P \sim p[\phi]$ which is the probabilistic analogue of the path quantifiers of CTL. For example, the PCTL formula P=?[F ϕ] states what is the probability of reaching state ϕ.

The complexity of PCTL query verification for DTMC is polynomial in the number of states [26]. Since the number of states of the DTMC is linear in the size of the program, overall complexity of PCTL query verification is polynomial of program size.

Loop Analysis In analyzing programs which contain back edges (either from loops or from recursion), I consider two different queries for programs with loops.

- What is the probability that target node n^t is reached at least once within a given loop bound?

- What is the probability that target node n^t is reached at least once?

The first query enables us to model bounded loop executions. To answer this query, I fix a loop bound and unroll any loops in the Markov chain. If the target node n^t is duplicated during this loop unrolling process, then the query becomes

- What is the probability that any target node n^t is reached at least once?

Once the loops in the Markov Chain are unrolled, the first query becomes the initial query on the unrolled Markov chain except that there might be multiple instances of the target node.

In answering the second query, I leave the Markov chain as is including any back edges and generate the DTMC model for PRISM as it is. PRISM calculates a steady state probability for unbounded loop scenario. Bounding the loop and asking the bounded version of the reachability query under approximates the unbounded case. As the loop bound increases, the solution for the bounded case approaches that of the unbounded case and in some cases it is possible to reach the steady state probability, i.e., to reach a fixpoint. Note that, in PRISM, I am able to compute the steady state probability, so it is not necessary to compute the fixpoint by increasing loop bounds. This is one of the advantages of my approach over probabilistic symbolic execution.

3.3 Implementation

I have implemented my technique in a tool called PReach (Probabilistic Reachability Analyzer) targeting programs written in Java programming language.

Using the static analysis tool Janalyzer [74] I first extract the control flow graph from the given program. After marking inputs for which I want to calculate reachability probability, I use dependency analysis for the marked inputs and identify all input-dependent branches. I identify the target statement node and do dominator and post-dominator analysis in order to extract the target statement subgraph.

For calculating branch selectivity of input-dependent branches I first translate the branch conditions to SMT-LIB format constraints using Spoon [77] and then I use ABC [46] for model counting. To compute refined branch selectivity I applied two abstract domains, interval and polyhedra using Jana [71], a numeric analysis tool for Java. I call these implementations as PReach-I and PReach-P respectively. I define the domain size for integers as signed 31 bit, for strings as length of 16 with all printable ASCII characters, for char as unsigned 8 bit integers. Once I get the model count from ABC, I calculate the branch selectivity. To compute bounded reachability of a target statement, I look for back-edges and if there is one, I unroll the loop to a certain bound. For unbounded cases, I compute the steady state probability.

Once I have all the branch selectivity values, I construct the discrete time Markov chain (DTMC). Using the target statement node, I formulate the queries to calculate the reachability probability. I use the probabilistic model checker PRISM [76] for computing the target statement reachability probability. I convert the Markov chain to a DTMC model in PRISM syntax and synthesize queries. Then, I execute PRISM to compute the probability. I use PRISM as it provides features to reduce the reachability checking of a statement in a program with unbounded loops to reachability checking of a state in DTMC. My current implementation determines reachability probability for each target statement separately. I can extend my approach to handle reachability of multiple statements by synthesizing slightly more complex queries.

For collecting ground truth values of *hard to reach* statements, I run a generator based random fuzzer for all the programs. I use JQF [67] tool which is a feedback directed fuzz testing platform for Java. JQF incorporates coverage-guided fuzz testing technique ZEST [68]. I use generator-based random fuzzing option provided by ZEST. I set a timeout of one hour and if the fuzzer fails to generate inputs to reach the target statement, I determine that the target statement is *hard to reach*.

3.4 Experimental Evaluation

To evaluate PReach, I experimented on benchmark programs from the Competition on Software Verification (SV-COMP) [23] and the Competition on Software Testing (Test-Comp) [66], which I call the SV-COMP benchmark. Recent competition (SV-COMP 2019) used 4 directories: MinePump, jayhorn-recursive, jbmc-regression and jpf-regression from the SV-COMP benchmark. So far, Test-Comp have only used C programs from the SV-COMP benchmark. Among the 4 directories used in SV-COMP 2019, I do not consider Minepump as the tasks are not dependent on the inputs. I use the other 3 directories and the algorithms directory for evaluation.

I mark all the non-deterministic inputs in the SV-COMP benchmarks as inputs for reachability analysis. I use the assert statements in these programs as target statements. I use two criteria to select the programs from these directories for my experiments. I exclude programs if one of the following two conditions hold:

1. Target statement reachability does not depend on the inputs: PReach is not applicable for these programs as it assesses reachability probability with respect to inputs.

2. Verification tasks are specific to floating point arithmetic: The model-counting constraint solver I use does not support constraints generated from such programs.

Based on the above criteria, my final dataset consists of a total of 142 programs. I modify these programs in order to allow us to run both my analysis and the generator based random fuzzer while keeping the program semantics unchanged. These modified programs are available at [78].

I run experiments on a virtual box equipped with an Intel Core i7-8750H CPU at 2.20GHz and 16 GB of RAM running Ubuntu Linux 18.04.3 LTS and the Java 8 Platform Standard Edition, version 1.8.0_232, from OpenJDK 64-Bit Server VM.

Results for the SV-COMP benchmark

Reachability probability computed by PREACH is a value between 0 and 1. In order to assess how good PREACH is to identify *hard to reach* statements, I classify program statements to two groups: *hard to reach* and *easy to reach*. As ground truth, I classify the programs for which the random fuzzer is unable to reach the target statement within the given time bound as *hard to reach*. I list the number of true positives (TP: ground truth is *hard to reach* and PREACH predicts *hard to reach*); false positives (FP: ground truth is *easy to reach* and PREACH predicts *hard to reach*); true negatives (TN: ground truth is *easy to reach* and PREACH predicts *easy to reach*); false negatives (FN: ground truth is *hard to reach* and PREACH predicts *easy to reach*). A *hard to reach* threshold (T_H) value 0.05 means statements having reachability probability less than 0.05 are classified as *hard to reach*. Then, I evaluate PREACH with respect to the ground truth.

Table 3.1 shows the overall precision, recall and accuracy results of PREACH-P. Precision, recall and accuracy for different implementations of PREACH is shown in Table 3.4. I demonstrate results for multiple values of T_H to analyze changes in precision, recall and accuracy across the benchmarks. Reducing T_H from 0.05 to 0.01 does not change the results at all. Increasing T_H to 0.1 leads to interesting changes in the results: some of the true negative cases are updated to false positives, reducing precision and accuracy. Increasing T_H to 0.25 changes the results further: the number of false positive cases are increased and number of true negative cases are decreased. Increasing the value of T_H changes the prediction of more cases from *easy to reach* to *hard to reach* and hence, the overall precision is reduced from 95.8% to 79.3% and the overall accuracy is reduced from 95.1% to 88.0%. The ability of using different threshold values demonstrates the quantitative nature of my analysis rather than being a fixed binary classification.

Accuracy of PREACH-P setting T_H as 0.05 or 0.01 is 95.1%. Across all the benchmarks, accuracy is greater than or equal to 87.0%, reflecting the effectiveness of my heuristic. PREACH-P fails to identify 5 of the *hard to reach* program statements having a recall of 90.2%, but it is very precise in identifying *hard to reach* program statements with a precision of 95.8%.

Among 142 cases, only 2 cases are false positives and 5 cases are false negatives. The remaining 135 cases are correctly classified by PREACH. The reasons behind the 2 false positive cases and the 5 false negative cases are: 1) most of the input values generated by the fuzzer lead to exceptions and the fuzzer cannot generate enough valid inputs, 2) the numeric analysis tool cannot handle complex operations such as multiplication, division and modulus between more than one variables using the abstract domains.

Experimental results show that among the 3 variations of the tools, PREACH-P performs the best with a precision, recall and accuracy of 95.8%, 90.2% and 95.1% respectively. Without applying refined branch selectivity, PREACH cannot catch two scenarios: 1) two dependent branch conditions cancel out each other, 2) input values are updated in a way that the branch condition becomes always true or false. Hence, the number of false negatives increases from 5 to 13. PREACH-I uses interval domain for refinement analysis which is not as precise as PREACH-P using a plyhedra domain. As a result, 2 extra false negatives are introduced by PREACH-I.

Table 3.1: Effectiveness of PREACH-P in terms of precision (Prec), recall (Rec) and accuracy (Acc) scores for sv-comp benchmarks

	Threshold (T_H)																				
Benchmarks	0.25							0.1							0.05/0.01						
	TP	FP	TN	FN	Prec	Rec	Acc	TP	FP	TN	FN	Prec	Rec	Acc	TP	FP	TN	FN	Prec	Rec	Acc
jayhorn-recursive	9	1	10	3	90.0	75.0	82.6	9	0	11	3	100.0	75.0	87.0	9	0	11	3	100.0	75.0	87.0
jpf-regression	25	7	43	2	78.1	92.6	88.3	25	2	48	2	92.6	92.6	94.8	25	2	48	2	92.6	92.6	94.8
jbmc-regression	8	1	12	0	88.8	100.0	100.0	8	0	13	0	100.0	100.0	100.0	8	0	13	0	100.0	100.0	100.0
algorithms	4	3	14	0	57.1	100.0	85.7	4	3	14	0	57.1	100.0	85.7	4	0	17	0	100.0	100.0	100.0
Total	46	12	79	5	79.3	90.2	88.0	46	5	86	5	90.2	90.2	93.0	46	2	89	5	95.8	90.2	95.1

Probabilistic Symbolic Execution (PSE)

I provide an experimental comparison of PReach with probabilistic symbolic execution (PSE) [19]. I use SPF [69] as the symbolic execution engine for PSE. PSE is unable to analyze some of the target programs due to unsupported constraints such as non-linear path constraints, PReach does not face this issue as much since it only considers branch conditions. The rest of the programs are marked as analyzable by PSE, as shown in Table 3.2. For programs where the number of recursive calls or loop iterations depend on the input, PSE can not explore all possible paths since it can only search programs behaviors up to a bounded execution depth (search depth), and since the number of program paths grows exponentially. Therefore, I set a timeout of 1 hour for PSE and evaluate for different search depths. Since PSE is unable to cover all program paths, the probabilistic measurement computed by PSE is not exact. Increasing the search depth allows PSE to obtain more accurate results but also increases the number of program paths exponentially. This leads PSE to time out for some programs, as shown in Table 3.2. This is not the case for the jpf-regression and jbmc-regression benchmarks, as there is no input dependent recursive calls or loops.

Table 3.2: Number of programs analyzed by PReach and Probabilistic Symbolic Execution within 1 hour timeout

Benchmarks	Number of programs analyzed								
	PReach	Probabilistic Symbolic Execution							
		Analy-zable	Analyzable with Search Depth						
			10	20	30	100	500	1000	∞
jayhorn-recursive	23	21	21	17	11	6	5	1	1
jpf-regression	77	69	69	69	69	69	69	69	69
jbmc-regression	21	16	16	16	16	16	16	16	16
algorithms	21	9	9	9	9	9	8	6	2
Total	142	115	115	111	105	100	98	92	88

I show the comparison of reachability probabilities computed by PReach and PSE in Table 3.3. As I do not have any ground truth for the probability measurement, I calculate probability differences between these two techniques and analyze the differences in case of agreement and disagreement for *hard to reach* statement assessment. PReach and PSE agree if their

Table 3.3: Probabilistic measurement differences and *hard to reach* statement prediction disagreements between PReach (PR) and PSE

Benchmarks	Search Depth	#Cases Analyzable	Tool	Agreement		Disgreement				All Cases Average Diff.
						PSE ✓		PReach ✓		
				#	Avg. Diff.	#	Avg. Diff.	#	Avg. Diff.	
jayhorn-recursive	10	21	PR	16	0.086	2	1.000	3	0.420	0.270
			PR-I	16	0.086	2	1.000	3	0.420	0.270
			PR-P	16	0.086	2	1.000	3	0.420	0.270
jpf-regression	∞	69	PR	58	0.050	10	0.550	1	0.250	0.083
			PR-I	62	0.049	6	0.542	1	0.250	0.095
			PR-P	64	0.035	4	0.625	1	0.250	0.072
jbmc-regression	∞	16	PR	14	0.040	2	0.250	0	-	0.066
			PR-I	16	0.031	0	-	0	-	0.031
			PR-P	16	0.031	0	-	0	-	0.031
algorithms	100	9	PR	3	0.087	0	-	6	0.390	0.317
			PR-I	3	0.087	0	-	6	0.390	0.317
			PR-P	3	0.087	0	-	6	0.390	0.317

predictions match, disagree otherwise. Based on agreement and disagreement, I divide all the cases into 3 groups: 1) agreement, 2) disagreement and PSE is correct, 3) disagreement and PReach is correct. The average difference in probability is low for the cases of agreement. The difference is even lower for jpf-regression and jbmc-regression benchmarks as PSE achieves very high precision and accuracy (see Table 3.5) and PReach agrees with the predictions. For the cases of disagreement, the difference is very high for most of the cases when PSE predicts correctly but PReach does not. One of the main reasons for this is variable updates making some of the program paths infeasible. PSE can catch the infeasible paths whereas PReach gives an approximate result for these cases using branch selectivity. Both PReach-I and PReach-P can address this issue. Using refined branch selectivity, the number of agreement cases are increased and average probability difference is reduced for jpf-regression and jbmc-regression benchmarks. Another reason is PReach predicting a program statement as *easy to reach* but the ground truth is *hard to reach* as fuzzer cannot reach the target statement due to recursion stack overflow error. Average difference is also high for jayhorn-recursive and algorithms benchmarks when PReach predicts correctly but PSE does not, as there is an exponential increase in the number of paths and PSE poorly approximates the probability.

I now compare these two techniques in terms of *hard to reach* statement prediction accuracy and precision. To compare PReach and PSE, I set the *hard to reach* threshold to 0.05. Table 3.4 shows precision, recall and accuracy for PReach and PSE with search depth 10 and 1000. I evaluate all 142 programs analyzable by PReach. The programs for which PSE times out are marked as *easy to reach* as my target is to identify the *hard to reach* program statements. Different search depths do not change results for jpf-regression and jbmc-regression benchmarks as these programs are free of recursive calls and loops that depend on inputs. The precision and accuracy values for PReach are comparable to PSE for these benchmarks. The prediction results are improved a lot using PReach-I and PReach-P. For jpf-regression and jbmc-regression benchmarks, precision, recall and accuracy are increased. For jbmc-regression benchmarks, both PReach-I and PReach-P performs better than PSE and for jpf-regression benchmarks, overall scores achieved by PReach-P are better than PReach-I and very close to the scores achieved by PSE. For jayhorn-recursive and algorithms benchmarks, PSE can not achieve as good results as PReach, PReach-I or PReach-P since these programs need to deal with input dependent recursive calls and loops. For lower search depth (10), PSE can not explore all the program paths and as a result the computed probability is an under-approximation (worse than a heuristics-based approach used in PReach). For higher search depth (1000), most of the programs time out and hence are marked as *easy to reach*. As a result there are no true-positive cases making precision and recall values 0 as well as no false-positive cases keeping the total precision high (96.9). For the algorithms benchmark, even with search depth 10, the precision and recall is 0 as PSE can not support most of the programs (marked as *easy to reach*) as array size is input dependent and marked as symbolic, which is not analyzable by SPF. Though for programs with bounded execution depth due to the absence of loop and recursion (jpf-regression and jbmc-regression benchmarks), PSE performs better than PReach but PReach-P is as good as or even better in some cases than PSE.

I show precision and accuracy for the 85 programs in these two benchmarks that are analyzable by PSE in Table 3.5. The scores for PSE are not 100% due to situations like integer arithmetic overflow that are not caught by symbolic execution. The precision (95.7) and accuracy (87.1) for PReach is comparable to PSE and is impressive given that it is a scalable heuristic approach. The precision (96.8) and accuracy (96.5) by PReach-P is very close to the scores achieved by PSE. Moreover, PSE performs very poorly on programs with unbounded execution depth (jayhorn-recursive and algorithms benchmarks) whereas PReach, PReach-I and PReach-P have high precision and accuracy.

Table 3.4: Precision, Recall and Accuracy of PReach (PR) and PSE, computed for 142 programs, program is marked *easy to reach* if analysis times out

Benchmarks	Precision					Recall					Accuracy				
	PR	PR-I	PR-P	PSE with Search Depth		PR	PR-I	PR-P	PSE with Search Depth		PR	PR-I	PR-P	PSE with Search Depth	
				10	1000				10	1000				10	1000
jayhorn-recursive	100.0	100.0	100.0	76.9	*0.0	75.0	75.0	75.0	83.3	*0.0	87.0	87.0	87.0	78.3	47.8
jpf-regression	90.5	92.0	92.6	96.2	96.2	70.4	85.2	92.6	96.2	96.2	87.0	92.2	94.8	97.4	97.4
jbmc-regression	100.0	100.0	100.0	100.0	100.0	75.0	100.0	100.0	75.0	75.0	90.5	100.0	100.0	90.5	90.5
algorithms	100.0	100.0	100.0	*0.0	*0.0	100.0	100.0	100.0	*0.0	*0.0	100.0	100.0	100.0	52.4	61.9
Total	95.0	95.7	95.8	80.4	96.9	74.5	86.3	90.2	82.0	62.0	89.4	93.7	95.1	79.6	85.9

Table 3.5: Precision, Recall and Accuracy of PSE and PReach (PR), out of 85 programs computed within 1 hour for jpf- and jbmc-regression benchmarks

Benchmarks	Precision				Recall				Accuracy			
	PR	PR-I	PR-P	PSE	PR	PR-I	PR-P	PSE	PR	PR-I	PR-P	PSE
jpf-regression	94.7	95.7	96.0	96.2	69.2	84.6	92.3	100.0	87.0	92.8	95.7	98.6
jbmc-regression	100.0	100.0	100.0	100.0	66.7	100.0	100.0	100.0	87.5	100.0	100.0	100.0
Total	95.7	96.6	96.8	96.9	68.8	87.5	93.8	100.0	87.1	94.1	96.5	98.8

Table 3.6 shows the average analysis time required and percentage of cases analyzed by both of these techniques. Even for a low search depth (10) the analysis time of PSE is higher than PReach. Note that, lower search depths in PSE poorly approximate the probability. However, increasing the search depth increases the analysis time by orders of magnitude. For both jayhorn-recursive and algorithms benchmarks, the average analysis time increases and percentage of analyzed cases within the time bound decreases as the search depth is increased. For the jayhorn-recursive benchmark, even for a search depth of 30 the average analysis time

increases by an order of magnitude. This is because the number of recursive function calls are input dependent. The average analysis time shown in the table is less than or equal to 3600 seconds since I set the timeout to 1 hour (i.e., 3600 seconds is the maximum analysis time). The time for jayhorn-recursive benchmarks with search depth greater than or equal to 30 would be very high without this timeout. Average analysis time also increases for the algorithms benchmarks when the search depth is increased as number of loop iterations depend on the inputs. These results show that PSE is not scalable for unbounded execution depth whereas PReach is.

PReach-I and PReach-P require more analysis time compared to PSE for jpf-regression and jbmc-regression benchmarks. As programs in these benchmarks are loop and recursion free, PSE runs fast whereas PReach-I and PReach-P perform abstract interpretation for branch selectivity refinement. However, as the search depth of the programs increases, the branch selectivity refinement analysis time becomes less significant compared to the exponential time increase due to path constraint solving performed by PSE, reflected in the jayhorn-recursive and algorithms benchmarks. For these benchmarks, as the search depth increases to 100, the analysis time by PSE is orders of magnitude higher than the analysis time required by PReach-I or PReach-P. These results clearly indicate that PReach, PReach-I and PReach-P maintain a balanced trade off between precision and scalability for probabilistic reachability analysis and among these three implementations, PReach-P performs the best considering its high precision and accuracy.

Statistical Symbolic Execution (SSE)

In this section, I provide an experimental comparison of PReach-P with statistical symbolic execution (SSE). Prior work has demonstrated that SSE is more precise and faster than PSE [21]. SSE also uses SPF [69] as the symbolic execution engine. I compare PReach-P and SSE only for the jayhorn-recursive and algorithms benchmarks from SV-COMP, as PSE

Table 3.6: Average Analysis Time for PREACH (PR) and PSE, maximum average analysis time is limited to 3600 seconds, cases with timeout are included

Benchmarks	Average Analysis time in seconds (% Cases Analyzed in 1 hour)						
	PR	PR-I	PR-P	Probabilistic Symbolic Execution Search Depth			
				10	30	100	1000
jayhorn-recusrive	2.43	4.35	6.16	5.34 (91%)	2048.33 (52%)	2583.22 (29%)	3428.67 (5%)
jpf-regression	0.81	3.11	4.86	1.51 (91%)	1.51 (91%)	1.5 (91%)	1.51 (91%)
jbmc-regression	0.69	4.90	6.10	3.32 (76%)	3.32 (76%)	3.32 (76%)	3.32 (76%)
algorithms	0.99	6.38	9.69	2.25 (43%)	3.98 (43%)	79.76 (43%)	2399.94 (29%)
Total	1.08	4.08	5.97	2.51 (82%)	372.50 (75%)	475.21 (71%)	808.28 (65%)

achieves very high precision and accuracy for jpf-regression and jbmc-regression benchmarks, and I have already compared the performance of PREACH-P and PSE on those benchmarks.

SSE is unable to analyze 12 out of 44 target programs due to inability to handle non-linear path constraints or symbolic array indexing during symbolic execution. As before, I set a timeout of 1 hour for SSE and evaluate for different search depths. Like PSE, SSE is also unable to explore all program paths within an hour, but it can provide statistical guarantees for the computed probabilities with respect to accuracy (ϵ) and confidence (δ) parameters [21]. SSE has two different sampling approaches: 1) Monte Carlo and 2) Informed sampling. I compare PREACH-P to both of these sampling techniques in SSE. In both cases, I set ϵ to be 10^{-5} and target δ to be 0.99 following the experimental setup in [21]. For Monte Carlo sampling, I set the maximum sample size (N_1) as $100,000$ and for informed sampling, I set N_1 as 100 and maximum number of iterations as 100.

Table 3.7: Precision, Recall and Accuracy of PREACH-P (PR-P) and SSE, computed for 44 programs from jayhorn-recursive and algorithms benchmarks, program is marked *easy to reach* if analysis times out(1 hour), both Monte Carlo and informed sampling has same precision, recall and accuracy

Benchmarks	Precision				Recall				Accuracy			
	PR-P	SSE with Search Depth			PR-P	SSE with Search Depth			PR-P	SSE with Search Depth		
		10	100	∞		10	100	∞		10	100	∞
jayhorn-recursive	100.0	100.0	100.0	100.0	75.0	75.0	75.0	58.3	87.0	87.0	87.0	78.3
algorithms	100.0	0.0*	0.0*	0.0*	100.0	0.0*	0.0*	0.0*	100.0	71.4	71.4	71.4
Total	100.0	100.0	100.0	100.0	83.3	50.0	50.0	38.9	93.2	79.5	79.5	75.0

Precision, recall and accuracy for SSE is presented in Table 3.7. SSE has better precision, recall and accuracy compared to PSE but not compared to PReach-P. Recall and accuracy for SSE drops with increasing search depth. For algorithms, precision and recall is 0.0 (marked with a *), as there were no true positive cases among the analyzable programs by SSE. Similar to the experimental setup for the comparison to PSE, I mark a program statement as *easy to reach* if it times out.

Table 3.8: Average Analysis Time and statistical Confidence (δ) for PReach-P (PR-P) and SSE Monte Carlo (MCS) and informed (IS) sampling, maximum average analysis time is limited to 3600 seconds, cases with timeout are included, confidence is set to 0.0 for timeout cases

Bench-marks	Average Analysis Time							Statistical Confidence (δ)					
	PR-P	SSE-MCS with Search Depth			SSE-IS with Search Depth			SSE-MCS with Search Depth			SSE-IS with Search Depth		
		10	100	∞	10	100	∞	10	100	∞	10	100	∞
jayhorn-recursive	6.16	1495.60	2117.46	2530.45	165.71	362.36	2038.48	0.061	0.039	0.032	0.957	0.913	0.435
algorithms	9.69	2558.30	3066.67	3071.77	2004.41	2008.00	2802.04	0.016	0.016	0.016	0.444	0.444	0.222
Total	7.84	2002.91	2577.38	2786.37	1046.43	1150.83	2406.93	0.040	0.028	0.025	0.712	0.690	0.333

I do not take the reported statistical confidence into account to determine which program statements should be marked as *hard to reach* or *easy to reach* by SSE. One could use a threshold value for the statistical confidence, and accept only the predictions achieving a certain confidence. In that case, the precision and accuracy of SSE would drop further. Instead, I present average confidence achieved by SSE in Table 3.8 separately. Statistical confidence achieved by SSE drops as the search depth for symbolic execution is increased and more programs time out. Even though I set a large maximum number of samples (100,000) for Monte Carlo sampling, SSE can not achieve a high confidence. On the other hand, informed sampling can achieve high confidence with search depths 10 or 100 for some cases. But, with an infinite search depth, none of the sampling techniques can achieve high confidence.

Average analysis time for SSE is presented in Table 3.8. In general, PReach-P is orders of magnitude faster than SSE. Monte Carlo sampling is consistently slower for all the programs compared to PReach-P. Informed sampling performs much better than Monte Carlo sampling.

Analysis time of SSE with informed sampling is close to PReach-P for some programs when a short search depth value is used. But, irrespective of search depth, for a good number of programs, informed sampling is also orders of magnitude slower than PReach-P, and hence its average analysis time is significantly higher than PReach.

These results demonstrate that PReach-P is more scalable compared to SSE and achieves better precision and accuracy, especially for programs containing large number of paths.

Case Studies

In this section, I evaluate the effectiveness of PReach to detect *hard to reach* program statements in larger projects. I am particularly interested in program points where inputs need to pass through numerous branches to reach. I selected a set of methods from Apache Commons Lang [24] and DARPA STAC Benchmarks [70] and identified target program statements. I have analyzed 24 program statements in 12 methods from Apache Commons Lang project and 12 program statements from 6 methods across 5 projects from DARPA STAC Benchmarks.

Table 3.9 shows PReach results for the selected 24 cases. First, I run PSE to compute reachability probability on all these cases. Among 18 methods I analyze I find that PSE is not able to handle 9 methods due to either variable type conversion or lack of support for some String library functions. PSE fails on 2 other methods due to incapability to model count for non-linear path constraints and another 4 methods due to lack of support for translation of expressions to string path constraints. PReach does not have any of these issues as the underlying technique is simpler than symbolically executing a program, and it can avoid dealing with non-linear path constraints and complex string path constraints as it needs to consider individual branch conditions only. Finally, PSE successfully runs on 3 methods but for 2 of the methods it times out, predicting only 1 case correctly as *hard to reach*. These results demonstrate the limitations and poor scalability of probabilistic symbolic execution on realistic programs. I also cannot analyze these cases using PReach-I and PReach-P as the programs perform string operations and the abstract interpretation tool [71] I use for computing refined branch selectivity is limited to numeric analysis. Even without refining the branch selectivity, my results for these case studies demonstrate that even the base technique (PReach) using branch selectivity is capable of predicting *hard to reach* program statements efficiently for sizable programs.

Table 3.9: Case study of PReach on Apache Commons Lang and DARPA STAC Benchmarks. PReach predicts a statement as *hard to reach* if reachability probability is less than 0.001. 19 out of 24 cases are predicted correctly

Project	Class Name	Method Name	Target Statement Line Number	Number of Branches in Method	Max #Branches to Target Statement	Reach Prob	Random Fuzzer Ground Truth	PReach Pred.	Pred. Match
apache-commons-lang	Fraction	greatest CommonDivisor	595	11	7	0.00%	Yes	Yes	✓
	NumberUtils	createNumber	759	31	25	0.00%	No	Yes	✗
		isCreatable	1690	25	23	0.33%	No	No	✓
	FastDatePrinter	parseToken	363	7	7	7.32%	No	No	✓
	StrTokenizer	readWithQuotes	804	8	8	0.00%	Yes	Yes	✓
	StrSubstitutor	substitute	837	17	13	0.00%	Yes	Yes	✓
	Numeric EntityUnescaper	translate	107	9	4	0.08%	No	Yes	✗
	ArrayUtils	shift	6994	9	9	0.00%	No	Yes	✗
	BooleanUtils	toBooleanObject	650	15	15	0.00%	Yes	Yes	✓
	RandomString Utils	random	427	16	16	0.00%	Yes	Yes	✓
	StringUtils	containsAny	1248	8	7	0.00%	Yes	Yes	✓
	CharSequence Utils	regionMatches	377	7	7	0.00%	Yes	Yes	✓
calculator_3	RomanNumeral Formatter	parseObject	130	34	17	0.15%	No	No	✓
			152		11	49.44%	No	No	✓
			170		34	0.59%	Yes	No	✗
calculator_4	Arithmetizer	assessParentheses	213	9	4	0.19%	No	No	✓
			245		9	93.75%	No	No	✓
	ImperialFormatter	parseObject	70	11	4	1.37%	No	No	✓
			96		9	24.63%	No	No	✓
			103		11	13.99%	No	No	✓
emu6502	Assembler	assembleLine	214	22	9	0.00%	No	Yes	✗
			207		8	0.15%	No	No	✓
linear_ algebra_ platform	MatrixSerializer	readMatrix FromCSV	51	13	9	7.30%	No	No	✓
rsa_commander	DecInputStream	read	99	19	12	0.08%	Yes	Yes	✓

PReach can predict 19 out of 24 cases correctly with an accuracy of 79.2% setting T_H as 0.001. I used the same value of T_H across all domains. Different values of T_H for Integer/mixed domain (0.01) and String domain (0.001) increases the accuracy to 83.33% supporting the quantitative nature of my analysis. 5 of the cases that PReach can not predict correctly is due to the similar reasons as SV-COMP benchmarks. The value of the input is updated inside the program and as a result the following branches do not depend on the initial input value anymore.

Chapter 4

Rare-path Guided Fuzzing

Testing software in order to assure its dependability and security is one of the most fundamental problems in software engineering. Fuzz testing has emerged as one of the effective testing techniques for achieving code coverage and finding bugs and vulnerabilities in software. Unfortunately, existing fuzzers often fail to generate inputs for program paths guarded by restrictive branch conditions. To pass through branch conditions most greybox fuzzers [1, 4, 79–82] focus on input mutation strategies. On the other hand, hybrid fuzzers [6, 83] switch to symbolic execution in order to solve path constraints when fuzzing gets stuck. Identifying the likelihood of the fuzzer getting stuck is a crucial problem for hybrid approaches, and using the fuzzer itself for this purpose (by monitoring fuzzing behavior) requires a lot of time to explore deeper paths.

Both input mutation-based fuzzers [4] and hybrid fuzzers [6] focus on identifying rare paths in the program to explore. They either use mutation strategies [4] or symbolic execution [6] to generate inputs that can explore the rare paths. Both of these techniques identify rare paths based on the inputs generated and branches covered during fuzzing (for example, AFL [1]). Note that, it may take a long time to generate a value that triggers a branch if the branch condition is very restrictive. It is difficult to separate infeasible paths from feasible but rare paths via input mutation.

In this paper, I propose a lightweight whitebox analysis to identify rare paths in programs and then guide symbolic execution to generate inputs to explore these rare paths. My approach avoids the shortcomings of mutation-based greybox fuzzers and hybrid fuzzers by generating inputs for rare paths beforehand, and it avoids the shortcomings of whitebox fuzzers based on symbolic execution by reducing the cost of symbolic analysis.

I present a heuristic for identifying rare paths where I use control flow analysis, dependency analysis and model counting on branch constraints to transform a control flow graph to a probabilistic control flow graph. Then, I compute path probabilities by traversing the probabilistic control flow graph and identify the rare (low-probability) paths.

To improve the rare path analysis I introduce a new type of control flow paths (which I call II-paths) which is a combination of intra- and inter-procedural program paths, providing a balance between breadth first and depth first traversal of program paths.

I guide concolic execution using rare paths to generate inputs that trigger these rare behaviors. As the last step of my approach, I provide the set of inputs from my analysis as the initial seed set to a fuzzer. This enables the fuzzer to explore the rare paths immediately, resulting in better coverage compared to randomly generated initial seed sets. My approach can be integrated with all existing fuzzers that rely on initial seeds.

My contributions in this paper are as follows:

- A new technique for identifying rare paths in programs using a lightweight quantitative symbolic analysis.
- A new type of control flow paths (II-paths) to improve efficiency and effectiveness of the rare path analysis.
- Algorithms for path-guided concolic execution.
- Rare-path guided fuzzing approach where the initial seed set for a given fuzzer is generated

with rare-path analysis.

- Experimental evaluation of the proposed techniques on existing fuzzers AFL++, FairFuzz and DigFuzz, demonstrating coverage improvement achieved by the proposed rare-path guided fuzzing approach.

Rest of the paper is organized as follows. In section 4.1, I provide overview of my technique. I explain program path analysis, heuristic to identify rare paths and input generation using the rare paths on section 4.2, 4.3 and 4.4 respectively. I discuss my implementation and experimental evaluations on section 4.5 and 4.6 respectively.

4.1 Overview

Consider the running example in Fig. 4.1 which is a shortened version of a code structure found in `libxml2`. The `main` procedure of the program reads a string as an argument. It checks if the first 3 characters of the string is `DOC` or not. If the first 3 characters of the string is `DOC`, it parses the string starting from the 4th character. First, it goes inside the `parse_cmt` procedure and it checks if the 4th character is `<` or `>` and skips if it is. Then, the program comes back to the `main` procedure and goes inside the `parse_att` procedure. In the `parse_att` procedure, the program looks for the character sequence `ATT`. If it finds this sequence, it goes deeper into the program and executes more functionalities. To summarize, the program is trying to find two specific sequences of characters: first `DOC` and then `ATT` and if it can find these two sequences, it can execute more functionalities.

A mutation based fuzzer, such as AFL, starting with a random initial seed will require a lot of mutations to get to an input containing sequences `DOC` and `ATT`. I run AFL 5 times on the running example for an hour. 4 out of 5 times, AFL cannot generate an input containing sequences `DOC` and `ATT`. AFL can generate inputs such as `DOC`, `DOC<`, `DOC>`, `DOCA` etc. Though coverage guided mutation helps to reach these inputs, AFL can not generate the desired sequences as it mutates randomly and breaks already found sequences to inputs likes `DAC` and `DOCQ` etc.

Now, let us explain how rare path analysis can guide a mutation-based fuzzer to achieve more coverage given a time budget. To perform rare path analysis on the running example program, I first extract the control flow graph and then I collect control flow paths of the program. At this point, I can use two well known existing techniques for control flow analysis to collect paths: intra-procedural control flow analysis and inter-procedural control flow analysis. Control flow graphs for the code in Fig. 4.1 are shown in Fig. 4.2.

First, I collect paths using intra-procedural control flow analysis (paths from 1 to 5 in Table 4.1). Among these paths, I find that path 4 is the rarest one. I identify rarity of the paths by computing path probability and I say that a path is the rarest if it has the lowest probability. Note that, to compute path probability, one can collect the path constraints using symbolic execution. In this paper, I do not use symbolic execution to collect path constraints. Instead I use a heuristic to compute path probabilities (discussed in section 4.3) that focuses on branch conditions and their selectivity.

After identifying the rare paths, I guide concolic execution (discussed in section 4.4) to generate inputs that trigger the rare paths. For example, for path 4 in Table 4.1, concolic execution generates the input `DOC`. I provide this input as the initial seed to AFL and I find that AFL can generate the sequences `DOC` and `ATT` within 40 minutes (on average) whereas AFL with a random seed cannot generate these sequence in an hour.

I also collect paths using inter-procedural control flow analysis (paths from 20 to 43 in Table 4.1). Using my rare path analysis, I identify path 35 as the rarest one. Guiding concolic execution using path 35, the input generated is DOC<ATT. Providing this input as initial seed, fuzzer immediately explores the path covering sequences `DOC` and `ATT`.

Using inter-procedural control flow analysis, I can generate the rarest paths in the program. However, paths based on inter-procedural analysis also traverse `parse_cmt` which is not necessary to generate the desired sequences `DOC` and `ATT` that enable us to explore deeper behaviors. Although, for my small running example, analyzing the procedure `parse_cmt` will not waste too much analysis time, for larger real world cases like `libxml2`, focusing only on inter-procedural paths is likely be costly and can increase the cost of rare path analysis significantly.

To improve the effectiveness of rare path analysis (in order to generate a higher number of rare seeds within a given time budget) I introduce a new kind of control flow path in this paper which I call II-paths (discussed in section 4.2). II-paths subsume intra-procedural and inter-procedural control flow paths, and include more paths that combine their characteristics. All the paths in Table 4.1 are II-paths, where paths 1 to 5 are intra-procedural control flow paths, and paths 20 to 43 are inter-procedural control flow paths. Furthermore, paths 6 to 19 are also II-paths. Let us assume that, given a time budget, I can generate the paths from 1 to 20 only. Then, I will identify II-path 13 as the rarest one and concolic execution can generate the input `DOCATT`. As a result, I will able to generate an input containing sequences `DOC` and `ATT` while analyzing a relatively small number of paths.

4.2 Program Paths

First step in rare-path guided fuzzing is identification of rare paths. The paths I identify are control flow paths that are generated by traversing control flow graphs of programs.

Control Flow Graphs

I define the control flow graph (CFG) [84] G_{proc} for a procedure *proc* as follows:

Definition 1 *A control flow graph for a procedure proc is a directed graph* $G_{proc} = (V, E)$ *where each vertex* $v \in V$ *represents a basic block of proc, and each directed edge* $e \in E : v \rightarrow v'$ *represents a possible flow of control from vertex* v *to vertex* $v' \in E$ *. Control flow graph* G_{proc} *has a unique entry vertex* $entry\_proc \in V$ *with no incoming edges and a unique exit*

```c
char *CUR;
#define CMP3( s, c1, c2, c3 ) \
  ( ((unsigned char *) s)[ 0 ] == c1 && \
    ((unsigned char *) s)[ 1 ] == c2 && \
    ((unsigned char *) s)[ 2 ] == c3 )
int main(int argc, char **argv) {
  CUR = argv[1];
  if (CMP3(CUR, 'D', 'O', 'C')) {
    CUR = CUR + 3;
    parse_cmt();
    if(parse_att())
      // go deeper
  }
  return 0;
}
void parse_cmt() {
  if(*CUR == '<' || CUR == '>')
    CUR++;
}
int parse_att() {
  if (CMP3(CUR, 'A', 'T', 'T'))
    return 1;
  return 0;
}
```

Figure 4.1: A code fragment based on the libxml file parser.c showing several nested branch conditions that must be satisfied to achieve higher code coverage.

vertex exit_proc $\in V$ with no outgoing edges. Furthermore, for each procedure call statement C to a procedure proc', G_{proc} contains a call vertex call-proc'$_C \in V$ and a return-site vertex return-proc'$_C \in V$, and an edge call-proc'$_C \rightarrow$ return-proc'$_C \in E$ that represents the procedure call.

Fig. 4.2 shows the control flow graphs for procedures shown in Fig. 4.1 in boxes a (`main`), b (`parse_cmt`) and c (`parse_att`).

Since programs typically contain multiple procedures, a single control flow graph can not represent the complete flow of control in a program. An inter-procedural control flow graph represents control flow of the whole program by combining the control flow graphs of all procedures of the program.

Definition 2 *An Inter-Procedural Control Flow Graph (IP-CFG) for a program P, $G_P^+ =$*

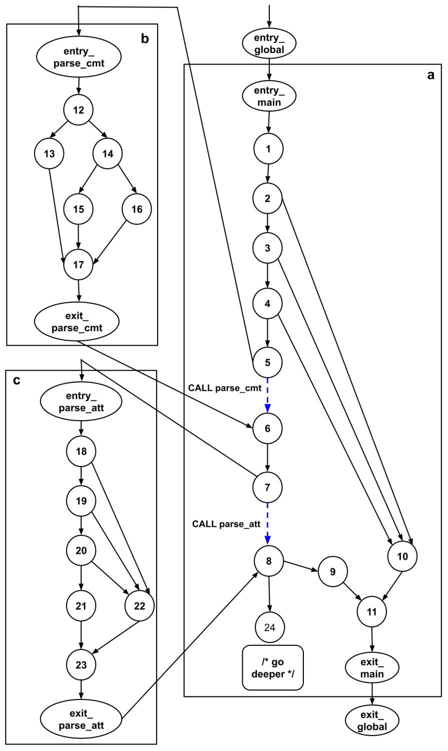


Figure 4.2: Inter-procedural control flow graph for the running example with additional edges for II-paths (marked in dashed blue line) and control flow graphs for `main`, `parse_cmt` and `parse_att` procedures (boxes a, b and c, respectively).

(V, E), contains the vertices and edges of the CFGs of all procedures in P, except the edges that correspond to procedure calls. Instead, for each procedure call statement C to a procedure proc in P, G_P^+ contains an edge from the call vertex to the entry vertex of the called procedure, call-proc$_C$ $\rightarrow$ entry_proc $\in E$, and an edge from the exit vertex of the called procedure to the return-site vertex for the call, exit_proc $\rightarrow$ return-proc$_C$ $\in E$, but it does not contain an edge between the call vertex and the return-site vertex, call-proc$_C$ $\rightarrow$ return-proc$_C$ $\notin E$. G_P^+ also contains a vertex entry_global $\in V$ with no incoming edges (entry point of the program) and another vertex exit_global $\in V$ with no outgoing edges (exit point of the program), and connects them to the main procedure of the program P with edges entry_global $\rightarrow$ entry_main $\in E$ and exit_main $\rightarrow$ exit_global $\in E$.

Fig. 4.2 shows the IP-CFG for my running example from Fig. 4.1 (the dashed edges are not part of the IP-CFG). For the call to procedure `parse_cmt`, there are two edges. One edge from call vertex 5 (which corresponds to *call-proc*$_C$) to *entry_parse_cmt* and one from *exit_parse_cmt* to return-site vertex 6 (which corresponds to *return-proc*$_C$). Similarly, for the call to procedure `parse_att`, there are two edges. One from call vertex 7 to *entry_parse_att* and one from *exit_parse_att* to return-site vertex 8. Note that, although in my running example in Fig. 4.1 each proecedure is called once, in general there could be multiple call statements for the same procedure and each call statement would have its own separate *call-proc*$_C$ and *return-proc*$_C$ vertices and corresponding edges in the IP-CFG.

Control Flow Paths

Now, I can define intra- and inter-procedural control flow paths based on the definitions above:

Definition 3 *Given a control flow graph* $G_{proc} = (V, E)$ *for a procedure proc, an intra-procedural control flow path (intra-path) is a sequence of vertices* $(v_1, v_2, v_3, \ldots, v_n)$ *where* $\forall i, v_i \in V, v_i \rightarrow v_{i+1} \in E$*,* $v_1 = entry\_proc$ *and* $v_n = exit\_proc$*.*

Definition 4 *Given an inter-procedural control flow graph* $G_P^+ = (V, E)$ *for a program P, an inter-procedural control flow path (inter-path) is a sequence of vertices* $(v_1, v_2, v_3, \ldots, v_n)$ *where* $\forall i, v_i \in V, v_i \rightarrow v_{i+1} \in E$*,* $v_1 = entry\_global$ *and* $v_n = exit\_global$*.*

Paths 1 to 5 in Table 4.1 correspond to all the intra-paths for the CFG of procedure *main*, and paths 20 to 43 in Table 4.1 are all the inter-paths for the IP-CFG of the whole program based on the control flow graphs shown in Fig. 4.2 for my running example. (To save space, I only show the vertices with numeric values in Table 4.1, named vertices that are not shown are implied by the sequence of vertices that are shown, and can be easily added.)

Intra-Inter Control Flow Paths (II-Paths)

In this paper, I introduce a new type of control flow paths by combining both intra-paths and inter-paths. I call these paths intra-inter control flow paths (II-paths). Intuitively, for each procedure call, inter-paths have to choose a path inside the called procedure's CFG. On the other hand, intra-paths do not explore the CFGs of the called procedures. When visiting a procedure call statement, II-paths have the option to either behave like intra-paths (i.e., do not explore the CFG of the called procedure), or behave like inter-paths (i.e., explore the CFG of the called procedure). Hence, II-paths have the option to ignore what happens inside a called procedure. The key insight is that, given the same length bound for inter-paths and II-paths, II-paths are able to explore a larger set of behaviors and deeper behaviors by ignoring the behavior of some called procedures.

In order to formally define II-paths I add back an extra edge to the IP-CFG between the call vertex $\textit{call-proc}_C$ and return-site vertex $\textit{return-proc}_C$ for each call statement C (as I had for the intra-procedural control flow graphs in Definition 1). I call the resulting control flow graph Extended Inter-Procedural Control Flow Graph (EIP-CFG):

Definition 5 *Let P be a program with IP-CFG $G_P^+ = (V, E)$. The Extended Inter-Procedural Control Flow Graph (EIP-CFG) for program P, denoted as $G_P^\star = (V', E')$, is defined as follows. The set of vertices for $G_P^\star$ is the same as the set of vertices for G_P^+, i.e., $V' = V$, and $G_P^\star$ contains all the edges in G_P^+, i.e., $E \subseteq E'$. The only edges that are in E' and not in E are: For each procedure call statement C, a single edge between the call vertex $\textit{call-proc}_C$ and the return-site vertex $\textit{return-proc}_C$ is included in E', i.e., $\textit{call-proc}_C \rightarrow \textit{return-proc}_C \in E'$ whereas $\textit{call-proc}_C \rightarrow \textit{return-proc}_C \notin E$.*

Fig. 4.2 shows the EIP-CFG for my running example from Fig. 4.1 where the dashed edges are also part of the EIP-CFG. In the EIP-CFG, there are two edges from each call vertex $\textit{call-proc}_C$ for a procedure call: 1) to the entry vertex of called procedure *proc entry_proc*, i.e., edge *call-proc*$_C$ → *entry_proc* and 2) to the return-site vertex *return-proc*$_C$, i.e., edge *call-proc*$_C$ → *return-proc*$_C$. For example, in Fig. 4.2, the call vertex 5 has two outgoing edges corresponding to these two cases 1) 5 → *entry_parse_cmt* and 2) 5 → 6. Similarly, call vertex 7 has two outgoing edges 1) 7 → *entry_parse_att* and 2) 7 → 8. As a result, whenever a call vertex is reached, there are two different paths to explore: 1) path taken via edge *call-proc*$_C$ → *entry_proc* which is similar to inter-paths, and 2) path taken via edge *call-proc*$_C$ → *return-proc*$_C$ which is similar to intra-paths. Intuitively, every time a procedure call vertex is reached, II-paths can choose between considering or ignoring the control flow inside the called procedure. Whereas, intra-paths never explore the control flow of called procedures, and inter-paths always have to explore the control flow of the called procedures.

I define II-paths as follows:

Definition 6 *Given an EIP-CFG $G^{\star}_P = (V, E)$ for a program P, an intra-inter control flow path (II-path) is a sequence of vertices $(v_1, v_2, v_3, \ldots, v_n)$ where $\forall i, v_i \in V, v_i \rightarrow v_{i+1} \in E$, $v_1 =$ entry_global and $v_n =$ exit_global.*

Again, let us consider the paths (listed in Table 4.1) of the EIP-CFG shown in Fig. 4.2 for my running example from Fig. 4.1. As I noted before, paths 1 to 5 in Table 4.1 are all the intra-paths for procedure *main*, and paths 20 to 43 in Table 4.1 are all the inter-paths for the program. Note that, based on the II-paths definition these paths are also II-paths. Furthermore, using the II-paths definition, in addition to II-paths from 1 to 5 and from 20 to 43, I now have additional II-paths from 6 to 19 where paths from 6 to 13 that ignore the control flow inside procedure `parse_cmt` but consider the control flow inside procedure `parse_att` and paths from 14 to 19 that ignore the control flow inside procedure `parse_att` but consider the control flow inside procedure `parse_cmt`.

Table 4.1: II-paths for the extended inter-procedural control flow graph shown in Fig. 4.2.

	Path	Probability
1	$1 \rightarrow 2 \rightarrow 10 \rightarrow 11$	9.96×10^{-1}
2	$1 \rightarrow 2 \rightarrow 3 \rightarrow 10 \rightarrow 11$	3.98×10^{-3}
3	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 10 \rightarrow 11$	1.59×10^{-5}
4	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11$	3.20×10^{-8}
5	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 24$	3.20×10^{-8}
6	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	3.19×10^{-8}
7	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	3.19×10^{-8}
8	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	1.27×10^{-10}
9	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	1.27×10^{-10}
10	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	5.10×10^{-13}
11	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	5.10×10^{-13}
12	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	2.05×10^{-15}
13	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 8 \rightarrow 24$	2.05×10^{-15}
14	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11$	1.28×10^{-10}
15	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 24$	1.28×10^{-10}
16	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11$	1.27×10^{-10}
17	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 24$	1.27×10^{-10}
18	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11$	3.17×10^{-8}
19	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 24$	3.17×10^{-8}
20	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	1.27×10^{-10}
21	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	1.27×10^{-10}
22	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	5.10×10^{-13}
23	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	5.10×10^{-13}
24	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	2.04×10^{-15}
25	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	2.04×10^{-15}
26	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	8.19×10^{-18}
27	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 13 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 8 \rightarrow 24$	8.19×10^{-18}
28	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	1.27×10^{-10}
29	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	1.27×10^{-10}
30	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	5.08×10^{-13}
31	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	5.08×10^{-13}
32	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	2.03×10^{-15}
33	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	2.03×10^{-15}
34	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	8.16×10^{-18}
35	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 15 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 8 \rightarrow 24$	8.16×10^{-18}
36	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	3.16×10^{-8}
37	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	3.16×10^{-8}
38	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	1.26×10^{-10}
39	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	1.26×10^{-10}
40	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	5.06×10^{-13}
41	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 22 \rightarrow 23 \rightarrow 8 \rightarrow 24$	5.06×10^{-13}
42	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 8 \rightarrow 9 \rightarrow 11$	2.03×10^{-15}
43	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 6 \rightarrow 7 \rightarrow 18 \rightarrow 19 \rightarrow 20 \rightarrow 21 \rightarrow 23 \rightarrow 8 \rightarrow 24$	2.03×10^{-15}

4.3 Identifying Rare Paths with Path Probability Estimation

To identify rare paths in the program, I first compute the probabilities for program paths. In this section, I explain construction of a probabilistic control flow graph to compute path probabilities. Then, I identify the rare paths in the program based on path probabilities.

Path Probability

I formalize path probability analysis as follows. Given a program P, let i denote the input for the program, and I denote the domain of inputs (i.e., $i \in I$). The input i for a program

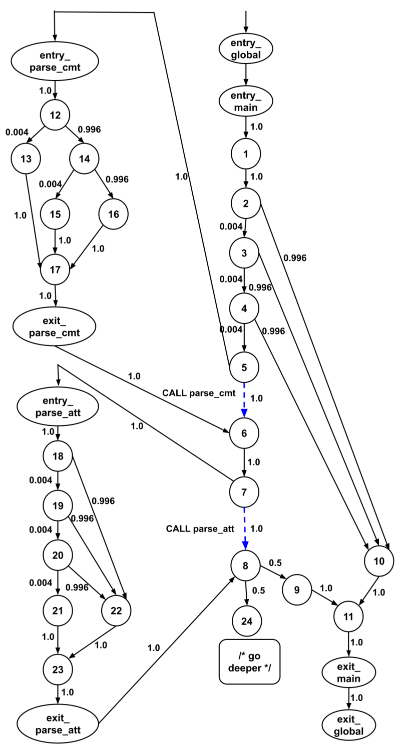


Figure 4.3: Probabilistic inter-procedural control flow graph corresponding to the inter-procedural control graph shown in Fig. 4.2, where edges are labeled with probability scores.

can be either a scalar value, a tuple, or a list of values. Given a path t in program P, the goal of path probability analysis is to determine how likely it is to execute the path t. I do this by determining the likelihood of picking inputs that result in an execution of path t. In order to determine the likelihood of picking such inputs, I compute the probability of picking such inputs given that the inputs are chosen randomly. I define $\mathcal{P}(P, t)$ as:

Definition 7 $\mathcal{P}(P, t)$ *denotes the probability of executing the path* t *of program* p *where the input* i *of the program* p *is randomly selected from the input domain* I *of program* p*.*

To compute path probability, I assume that inputs are uniformly distributed. However, one can extend my technique for path probability computation by integrating usage profile [21], used in other probabilistic analysis techniques and support any input distribution.

Path probabilities can be computed using quantitative extensions of symbolic execution such as probabilistic and statistical symbolic execution [17, 18]. However, these symbolic execution based techniques have a high computation complexity and poor scalibility due to the cost of path constraint solving and model counting over an exponentially increasing number of paths. Recently, there has been a new heuristic-based technique proposed for probabilistic reachability analysis [22]), which reduces the complexity of probabilistic reachability analysis using branch selectivity. The heuristic in [22] focuses on computing the reachability probabilities for program statements using a concept called branch selectivity. In this paper, I also use branch selectivity to estimate path probabilities. Note that, my goal is not to analyze probabilistic reachability of a target statement as in [22]. I focus on estimating path probabilities.

Probabilistic Control Flow Graph

To compute path probabilities, I construct a probabilistic control flow graph (Prob-CFG) $PG^{\star}_{P}$ for a program P from the extended inter-procedural control flow graph (EIP-CFG) $G^{\star}_{P}$. I formally define the probabilistic control flow graph $PG^{\star}_{P}$ as follows:

Definition 8 *Given a program P and its EIP-CFG $G^{\star}_{P} = (V, E)$, the probabilistic control flow graph $PG^{\star}_{P}$ for program P is defined as $PG^{\star}_{P} = (V, E, F)$ where the set of vertices and edges for $PG^{\star}_{P}$ are same as the set of vertices and edges of $G^{\star}_{P}$, and F is a function $F : E \rightarrow [0, 1]$ that assigns a probability score to each edge in E.*

As I describe below, I use dependency analysis and branch selectivity to compute probability scores of the edges in probabilistic control flow graphs.

Dependency Analysis A branch condition in the program is input dependent if the evaluation of the branch condition depends on the value of the program input. Given a program and input(s) to the program, I use static dependency analysis to identify the input dependent branch vertices in the control flow graph. Static dependency analysis over-approximates the set of input-dependent branch vertices. As a result, the path probability I compute is an estimation of the actual path probability. Anyway, I use branch selectivity, a heuristic to estimate path probability.

Branch Selectivity To compute the probability for each edge in the control flow graph, I use branch selectivity. I use the definition of branch selectivity $\mathcal{S}(b)$ as in [22]:

Definition 9 *Given a branch condition b, let D_b denote the Cartesian product of the domains of the variables that appear in b, and let $T_b \subseteq D_b$ denote the set of values for which branch condition b evaluates to true. Let $|D_b|$ and $|T_b|$ denote the number of elements in these sets, respectively. Then, $\mathcal{S}(b) = \frac{|T_b|}{|D_b|}$ and $0 \leq \mathcal{S}(b) \leq 1$.*

I compute $|T_b|$ using a model counting constraint solver. Branch selectivity gets closer to 0 as the number of values that satisfy the branch condition decreases and gets closer to 1 as the number of values that satisfy the branch condition increases.

Then, I define the probability score function F for the probabilistic control flow graph $PG_P = (V, E, F)$ using the combination of dependency analysis and branch selectivity as follows:

- If there is only one edge starting from a vertex v to u, then the probability of the edge $e : v \to u$ is 1, i.e, $F(e) = 1$.
- If v is a vertex with branch condition b, there are two edges from source vertex v: $e_1 : v \to u_1$ and $e_2 : v \to u_2$, where e_1 is the true evaluation and e_2 is the false evaluation of branch condition b:
    - If branch condition b is dependent on program input, then probability of edge e_1 is the branch selectivity, $\mathcal{S}(b) = \frac{|T_b|}{|D_b|}$ and the probability of edge e_2 is $1 - \mathcal{S}(b)$, i.e., $F(e_1) = \mathcal{S}(b)$ and $F(e_2) = 1 - \mathcal{S}(b)$.

  - If branch condition b is not dependent on program input, then probability of both edges e_1 and e_2 is 1, i.e., $F(e_1) = F(e_2) = 1$.

- Probabilities of edges that have a call vertex as their source e_1 : *call-proc*$_C$ $\to$ *entry_proc* and e_2 : *call-proc*$_C$ $\to$ *return-proc*$_C$ are 1, i.e., $F(e_1) = F(e_2) = 1$.

By adding probabilities to all the edges in a control flow graph, I transform it to a probabilistic control flow graph. Consider the EIP-CFG in Fig. 4.2. Each branch vertex is associated with a branch condition. For example, vertex 2 is associated with branch condition `CUR[0] = D`. I consider that the inputs are uniformly distributed and domain for each character in a string has 256 values. Branch selectivity $\mathcal{S}$ for the branch condition at vertex 2 is $\frac{1}{256} \equiv 0.004$. Hence, probability for the edges $2 \to 3$ is 0.004 and probability for the edge $2 \to 10$ is $1 - 0.004 = 0.996$. I add all the edge probabilities to the EIP-CFG $G^{\star}_P$ in Fig. 4.2 and construct the probabilistic EIP-CFG $PG^{\star}_P$, shown in Fig. 4.3.

Once I construct the probabilistic control flow graph $PG^{\star}_P$, I can compute path probabilities by traversing the graph. The path probability I compute for a path is the multiplication of the probabilities of the edges on the path. The path probability I compute is also a real number between 0 and 1, where a path probability close to 1 means that the path is executed by almost every input and a path probability close to 0 means that the path is executed by very few inputs. I compute path probability $\mathcal{P}(P, t)$ in PG_P as follows.

Definition 10 *Given a control flow path t for program P which corresponds to a sequence of vertices $\{v_1, v_2, v_3, \ldots, v_n\}$ in the probabilistic control flow graph $PG^{\star}_P = (V, E, F)$, then path probability $\mathcal{P}(P, t)$ for path t is computed as*

$$\mathcal{P}(P, t) = \prod_{i=1}^{n-1} F(v_i, v_{i+1})$$

Path probabilities computed for the II-paths for my running example using the probabilistic control flow graph in Fig. 4.3 are shown in Table 4.1.

Rare Paths

My idea behind rare paths are that some paths in the program are executed less frequently compared to any other paths in the program. If any input i is picked randomly from an uniformly distributed domain of inputs I, it is less likely for a rare program path being executed by input i compared to the execution of any other paths in the program. I define a set of rare paths R_p as follows:

Definition 11 *Given a set of n paths $A = \{t_1, t_2, t_3, ..., t_n\}$ and the set of k rare paths $R = \{u_1, u_2, u_3, ..., u_k\}$ in program P, where $R \subset A$, probability for any path in R will be less than probability for all paths in $A \backslash R$, i.e., $\forall t, u, t \in A \backslash R \wedge u \in R \Rightarrow \mathcal{P}(P, u) \leq \mathcal{P}(P, t)$.*

First, I compute probability for the set of paths A_t and then I sort A_t in an ascending order based on the probabilities of the paths. After that, I select first k paths as the set of k rare paths R_t.

Traversing through the probabilistic control flow graph in Fig. 4.2 I generate 43 II-paths and compute corresponding path probabilities as shown in Table 4.1. Now, if I sort these paths in an ascending order based on the path probability and pick the set of rare paths R for $k = 3$, I identify paths 34, 35 and 26 as the paths in the rare path set R. A fuzzer that randomly generates inputs would be very unlikely to explore these rare paths.

4.4 Input Generation for Rare Paths

The analysis I described above results in the set R of k rare paths in the program. However, it does not identify k inputs that can trigger these rare paths in the program. The input generation process I describe in this section identifies inputs to trigger the rare paths in the set R.

In order to generate the set of rare inputs I_R for the set of rare paths R I guide concolic execution using each rare path $t_R \in R$ and generate input i_R for each t_R (if path t_R is a feasible execution path). I add all these inputs to the set of rare inputs I_R.

Note that, the rare paths I compute are based on an estimation of path probability and some of the rare paths might not be feasible (not a real execution path in the program). But, concolic execution captures the original program execution semantics. Hence, if a rare path is not feasible, it will be detected in the input generation step using concolic execution.

I use path-guided concolic execution to collect path constraints for a rare path. I use a SMT solver to solve the path constraints and generate the input that can be fed to the program to execute the rare path. I will now discuss the mechanism in detail for path-guided concolic execution.

In section 4.2, I have discussed three different types of control flow paths: intra-paths, inter-paths and II-paths. Out of these three types of paths, inter-paths always represent complete program execution paths. However, both intra-paths and II-paths represent either partial or complete program execution paths as these two types of paths ignore procedure calls either completely (intra-paths) or as a choice (II-paths). I provide two different algorithms for path-guided concolic execution for input generation: 1) Inter-path guided concolic execution, 2) Intra-path or II-path guided concolic execution.

Inter-path guided Concolic Execution (IP-GCE)

For inter-path guided concolic execution (IP-GCE), I run the program on a concrete random input and generate the corresponding inter-path t_C. In order to generate input for the rare path t_R, I compare all branches for t_C and t_R in the same order. If there is a mismatch between any of the branches, I negate the branch and solve it to check feasibility of the path negating the branch. If the path is feasible, I solve the path constraint and generate the new input. I then execute the program using the new input and update t_C by the inter-path generated by the new input. The process continues as long as there are branches left to compare both in t_C and t_R or there are no branches that can lead to a feasible path. At the end of the process, the input is the input that will either take path t_R or take a path that is close to the rare path t_R if t_R is not feasible.

Algorithm 2 shows the process of guiding concolic execution using rare inter-path. Execute executes the program P first on a random input and returns the corresponding execution path t_C. The algorithm looks for the first vertex where t_C and t_R differ (all paths start with the same vertex). NegatedPath(t_C, *index*) generates a path constraint corresponding to the path t_C where the branch condition between the vertex $index - 1$ and *index* is negated. IsFeasible checks the feasibility of a given path constraint and Solve generates an input value satisfying the given path constraint.

Algorithm 2 IP-GCE(P, t_R)
Takes a program P and an inter-procedural path t_R in P as input and generates an input for P to execute the path t_R

```
 1: $input \leftarrow$ Random()
 2: $t_C \leftarrow$ Execute($P, input$)
 3: $index \leftarrow 2$
 4: while $index <$ Len($t_C$) $\wedge$ $index <$ Len($t_R$) do
 5:     if $t_C(index) \neq t_R(index)$ then
 6:         $path\_cond \leftarrow$ NegatedPath($t_C, index$)
 7:         if IsFeasible($path\_cond$) then
 8:             $input \leftarrow$ Solve($path\_cond$)
 9:             $t_C \leftarrow$ Execute($P, input$)
10:         else
11:             return $input$
12:     $index \leftarrow index + 1$
13: return $input$
```

II-Path Guided Concolic Execution (IIP-GCE)

In this section I discuss II-Path guided concolic execution which can also handle intra-paths since intra-paths are also II-paths. IP-GCE algorithm I discussed in the previous section uses branch matching and branch negation for mismatched branches, but this approach is not sufficient for guiding the concolic execution to explore the rare II-paths since II-paths are not guaranteed to represent complete execution path of a program.

Similar to the *IP-GCE* algorithm, in the *IIP-GCE* algorithm (Algorithm 3), I first run the program on a concrete random input and collect the execution path t_C. Note that, there may be branches in t_C that are in a procedure that is not explored in the input II-path t_R. In such situations, I compare the inputs that trigger both the branch and its negation, and see which one creates a path that overlaps more with t_R (i.e., the number of vertices that are common in both), and then I pick the branch which results in higher overlap with t_R. For branches that appear both in t_C and t_R I make sure that t_C and t_R agree.

Lines 1-5 in Algorithm 3 generate the initial concrete path t_C with a random input, and calculate the initial overlap between t_C and t_R using the function Overlap.

The while loop in lines 6-19 iterates over the nodes in t_C. It looks for branch nodes in t_C that differ from the corresponding branch node in t_R. The function Differ returns true under two conditions: 1) there is a branch in t_R that corresponds to complement of $t_C(\mathit{index})$ (i.e., t_R and t_C take different branches for the same branch statement), or 2) there is no branch in t_R that corresponds to the branch $t_C(\mathit{index})$ In both of these cases I negate the branch condition at $t_C(\mathit{index})$ and see if I can improve the overlap between t_C and t_R and update the input and t_C if the overlap can be improved. Note that if the overlap cannot be improved the input is restored to the previous input in lines 17-18.

Algorithm 3 makes a single pass on t_C without backtracking and therefore it is not guaranteed to find an execution that maximizes the overlap between final t_C and t_R. Looking for maximum overlap would require a search on all execution paths, resulting in path explosion that I have to avoid for scalability.

Algorithm 3 IIP-GCE(P, t_R)
Takes a program P and a intra- or II-path t_R in P as input and generates an input for P to execute the path t_R

```
1: $input \leftarrow$ RANDOM()
2: $t_C \leftarrow$ EXECUTE($P, input$)
3: $max\_overlap \leftarrow$ OVERLAP($t_C, t_R$)
4: $max\_input \leftarrow input$
5: $index \leftarrow 1$
6: while $index <$ LEN($t_C$) do
7:     if ISBRANCH($t_C(index)$) $\wedge$ DIFFER($t_C(index), t_R$) then
8:         $path\_cond \leftarrow$ NEGATEDPATH($t_C, index$)
9:         if ISFEASIBLE($path\_cond$) then
10:            $input \leftarrow$ SOLVE($path\_cond$)
11:            $t_C \leftarrow$ EXECUTE($P, input$)
12:            $overlap \leftarrow$ OVERLAP($t_C, t_R$)
13:            if $overlap > max\_overlap$ then
14:                $max\_overlap \leftarrow overlap$
15:                $max\_input \leftarrow input$
16:            else
17:                $input \leftarrow max\_input$
18:                $t_C \leftarrow$ EXECUTE($P, input$)
19:     $index \leftarrow index + 1$
20: return $input$
```

For the running example, guiding concolic execution using path 35, input generated is `DOC<ATT`. whereas guiding concolic execution using path 34, I find out that path 34 is infeasible. Path 34 is infeasible as path up to vertex 8 in path 34, $parse\_att$ function returns 1 and then returning back to the main function it should take the path following edge $8 \rightarrow 24$ whereas it takes edge $8 \rightarrow 9$. Hence, path-guided concolic execution algorithms I provide does not only generate inputs but also checks feasibility of the rare paths. Even though my techniques for identifying rare paths in the program is a heuristic approach, infeasible rare paths will be always filtered out in the input generation phase. The inputs I provide to the fuzzer as initial seed sets are always valid inputs and they help fuzzer to explore deep rare program paths.

4.5 Implementation

I implement my techniques for rare path analysis and path-guided concolic execution targeting programs written in C programming language.

I extract branch conditions and control flow graph for a program using the concolic execution tool CREST [15] and underlying program transformation tool CIL [85].

In order to collect branch conditions from the program, I modify the OCaml code in CIL. I transform the branch conditions in the program to constraints in SMT-LIB format. To model count the branch constraints, I use Automata-based Model Counter (ABC) [46].

To identify input dependent branches in the program, I do dependency analysis using CodeQL [86] code analysis engine. CodeQL treats code like data and can run queries on any codebase hosted in the GitHub. To implement dependency analysis, I use Access module of CodeQL that provides classes for modeling accesses including variable accesses, enum constant accesses and function accesses.

After extracting the control flow graph and model counting the input dependent branches, I transform the control flow graph to a probabilistic control flow graph. I use a python script to traverse the probabilistic control flow graph and collect paths. I wrote 3 different functions in python to collect intra-paths, inter-paths and II-paths.

I guide concolic execution tool CREST [15] using the rare paths I collect from my control flow analysis. I wrote algorithms IP-GCE and IIIP-GCE in C on top of existing concolic search strategies in CREST.

I use existing coverage-guided fuzzers AFL++ [87] and FairFuzz [4] as it is. I could not find implementation of DigFuzz [6]. I contacted the authors of DigFuzz but it was not publicly available at the time of my implementation. I implement DigFuzz using AFL++ and QSym [88]. To collect edge coverage I use `afl-showmap` as used in [89].

4.6 Experimental Evaluation

To evaluate my techniques for rare path-guided fuzzing I experiment on a set of benchmarks (programs with many restrictive branch conditions) that have already been used in experimental evaluation of existing fuzzing techniques. *inih* (parser for .ini configuration file), *tinyC* (parser for tiny C codes with if-else, while, do-while structures), *cJSON* (parser for JSON files) have been used for evaluating parser-directed fuzzing [90]. I also add *calculator* [91] (a command-line calculator, supporting standard mathematical operations and a set of function), more complex in terms of restrictive branch conditions. I also experiment on two well known libraries for parsing xslt and xml files, *libxslt* and *libxml2* respectively. *libxslt* has been used in [92] and *libxml2* has been used to evaluate many coverage guided fuzzing techniques [1, 4, 79].

In my experimental evaluation I focused on the following research questions:

RQ1. Can rare path analysis generate inputs that AFL++ can not?

RQ2. Can I improve fuzzing effectiveness using the seed set I generate from my rare path analysis?

RQ3. Can I improve rare path analysis effectiveness using II-paths?

Experimental Setup

I run my experiments on a virtual box equipped with an Intel Core i7-8750H CPU at 2.20GHz and 16 GB of RAM running Ubuntu Linux 18.04.3 LTS. I use docker for AFL++ [93] to run all the fuzzing experiments. I run each fuzzing task with a random seed set for 24 hours. I set the upper limit for my rare path guidance technique (branch selectivity computation, rare path identification and seed generation) to 6 hours (25% of the total time) and use the remaining 18 hours (of 24 hour total time) fuzzing with the seed set generated by my analysis. I set path depth limit to 60 for my rare path analysis. After collecting the rare paths, I provide all the inputs from the feasible rare paths (filtered by path-guided concolic execution) to the fuzzer as the seed set.

Experimental Results

RQ1: Effectiveness of rare path analysis to generate rare inputs

To show the effectiveness of my rare path analysis, I run my analysis maximum for 6 hours and AFL++ for 24 hours on each of these benchmarks. My experimental results show that I can generate inputs in 6 hours which AFL++ cannot generate in 24 hours by mutating inputs. My results in detail are as follows.

tinyC. I generate inputs containing *if-else* structure from my rare path analysis. AFL++ can generate *if* structure by mutating inputs but cannot generate the *if-else* structure.

inih. Each ini file has section names inside an opening bracket, [and a closing bracket,] and key value pairs separated by either a colon (:) or an equal sign (=). From my rare path analysis I can find these rare input structures within a minute. But, AFL++ can also generate these inputs within couple of minutes as the input structure is trivial. So, for *inih*, I cannot generate any new inputs.

calculator. I generate inputs containing keywords such as *arcsin*, *arccos* and *arctan* with my rare path analysis. Even after running AFL++ for 24 hours, AFL++ cannot generate these keywords.

cJSON. AFL++ can generate inputs containing basic JSON structure with left and right braces, colon and quotations. But, using my rare path analysis, I can generate inputs containing keywords such as *false*, *true* and *null* that AFL++ is unable to generate.

libxslt. To explore deeper paths in the program xslt files need to contain keywords like `stylesheet`, `transform`, `attribute-set`, `preserve-space`, `decimal-format` etc. As a random seed, I provide XSLT file containing opening and closing tag for `stylesheet` to AFL++. However, running AFL++ for 24 hours, it cannot generate inputs containing any other keywords. My rare path analysis can generate inputs containing keywords: `attribute-set`, `preserve-space` and `decimal-format`.

libxml2. Similar to *libxslt*, to explore deeper paths in *libxml2*, a xml file needs to contain keywords like `DOCTYPE`, `ATTLIST`, `ENTITY`, `NOTATION` etc. Running AFL++ for 24 hours, it can generate inputs containing structures like `DOCTYPE` and `ATTLIST`. My rare path analysis can generate inputs containing not only `DOCTYPE` and `ATTLIST` but also `ENTITY` and `NOTATION`.

Overall, I see that for 5 out 6 benchmarks, within 6 hours (25% of the time allocated to AFL++), my rare path analysis can generate inputs that AFL++ cannot generate in 24 hours based on input mutation.

RQ2: Effectiveness of rare path analysis to improve fuzzing effectiveness

My answer to RQ1 already shows that the rare path analysis can generate inputs that AFL++ cannot. Now, to answer RQ2, I present experimental results evaluating the ability of rare path analysis in improving fuzzing effectiveness in terms of coverage.

My experimental results show that (as shown in Fig. 4.4 and Table 4.2) I get coverage improvement over AFL++ for 5 out of 6 of the benchmarks. I do not get a lot of improvement for *calculator* (1.13%) since, even though I can generate rare inputs, there are no deeper functionalities to execute after passing through the rare branches. I generate inputs containing functions: *arcsin*, *arccos*, *arctan* using my rare path analysis. And with these additional inputs, AFL++ can mutate and generate 3 more rare inputs: *asin*, *acos*, *atan*. However, there are not many functionalities to explore and code to cover after these rare branches. AFL++ with the rare path based seed set can cover only 13 additional edges (1.33% improvement). For *tinyC* and *cJSON*, I see improvement of 6.47% (13 additional edges) and 4.19% (25 additional edges), respectively. For *libxslt*, my rare path guidance helps AFL++ to cover 162 additional edges (18.86% coverage improvement). For *libxml2*, I achieve the maximum amount of coverage improvement of 1170 additional edges (20.35%). This indicates that for larger programs if restrictive branches in the

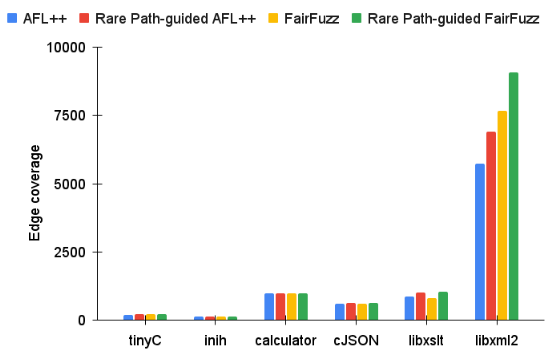


Figure 4.4: Coverage comparison between AFL++, rare-path guided AFL++, FairFuzz and rare-path guided FairFuzz

program can be passed, fuzzers can explore deeper functionalities and achieve significantly more code coverage, and my rare path analysis can guide the fuzzers to pass the restrictive branches in the program.

Next, I experimentally evaluate my rare path analysis using FairFuzz [4] using the same setup that I used for AFL++. For 5 out of 6 cases, I see improvement, 0.51% for *calculator* 0.94% for *tinyC*, 4.14% for *cJSON* and 18.29% for *libxml2* (shown in Fig. 4.4 and Table 4.2). The results are similar to AFL++, for larger programs, FairFuzz can explore more deeper functionalities and achieve more code coverage. For *libxslt*, FairFuzz without any guidance can cover 800 edges whereas with guidance it can cover 1055 edges (31.86% coverage improvement). For *libxml2*, FairFuzz without inputs from my analysis can cover 7681 edges, whereas with guidance from rare path analysis, it can cover 9086 edges (18.29% of coverage improvement).

Moreover, for *cJSON*, *libxslt* and *libxml2*, my rare path analysis can generate inputs that FairFuzz cannot. This indicates that FairFuzz (which uses branch hit counts to identify rare branches) can not pass some rare branches. However, I can identify and generate inputs for these rare branches. Rare path guided FairFuzz performs best in my experimental evaluation (1.33%, 5.36%, 7.46%, 22.82% and 58.00% more coverage than AFL++ for *calculator*, *cJSON*, *tinyC*, *libxslt* and *libxml2* respectively).

Table 4.2: Percentages of coverage improvement for rare path-guided fuzzing over AFL++, FairFuzz

Benchmarks	Number of lines	% coverage improvement over	
		AFL++	FairFuzz
tinyC	190	6.47%	0.94%
inih	243	0.00%	0.00%
calculator	1312	1.33%	0.51
cJSON	3845	4.19%	4.14%
libxslt	33371	18.86%	31.86%
libxml2	186116	20.35%	18.29%

Lastly, I evaluate my rare path analysis on top of hybrid fuzzing technique, DigFuzz [6]. DigFuzz [6] identifies the hardest paths to explore for AFL using the samples collected using AFL and then uses symbolic execution tool angr [94] to solve constraints for the hardest paths. However, DigFuzz is not publicly available. I contacted the authors of DigFuzz but could not get access to the implementation. Hence, I implement the technique in DigFuzz using AFL++ and QSym [88]. In my evaluation I use an unoptimized binary for fuzzing (to associate branch flip in concolic execution with hitcount collected in fuzzing which is necessary for the implementation of the DigFuzz technique). I conduct experiments on the 3 larger benchmarks, *cJSON*, *libxslt* and *libxml2*. Results from my experimental evaluation (Table 4.3) show that rare path guided DigFuzz achieves better coverage compared to DigFuzz, 66.86% improvement for *cJSON*, 2.18% improvement for *libxslt* and 30.22% improvement for *libxml2*.

There are multiple reasons behind DigFuzz not being able to achieve better coverage compared to AFL++ and FairFuzz: 1) building the execution tree takes hours for larger programs like *libxml2* as the tree grows exponentially over time, 2) concolic execution fails to generate inputs for a lot of paths and hence generates very few inputs to guide AFL and 3) DigFuzz attempts to solve branches that are not dependent on the inputs rather used for sanity check of the program. These findings are aligned to the findings of DigFuzz for larger programs [6]. However, my experiments on DigFuzz still demonstrate that rare path guided analysis improves the effectiveness of DigFuzz like it improves AFL++ and FairFuzz.

Table 4.3: Percentages of coverage improvement for rare path-guided fuzzing over DigFuzz

Benchmarks	DigFuzz	Rare Path-guided DigFuzz	% coverage improvement
cJSON	344	574	66.86%
libxslt	719	735	2.18%
libxml2	3297	4270	30.22%

RQ3: Effectiveness of II-path to improve efficiency of rare path analysis

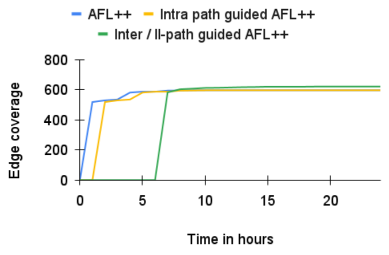
To answer RQ2, I guide fuzzers using my rare path analysis based on intra-paths, inter-paths and II-paths. My claim is that II-paths based analysis can generate more rare inputs compared to either intra-paths or inter-paths or both. My experimental results for *cJSON*, *libxslt* and *libxml2* is shown in (Fig. 4.5) respectively.

Using intra paths for *cJSON*, I do not see any improvements as I cannot generate any inputs. However, using inter paths I can generate inputs and see improvements (4.19%). Using II-paths I can also generate the same inputs and see same amount of coverage improvement.

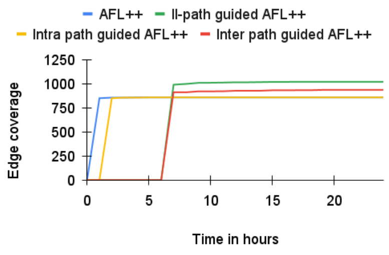
For *libxslt*, using intra paths, I do not see any improvement as it can can not generate new inputs. Using inter paths, I see coverage improvement (9.08%) as new inputs are generated containing keywords `preserve-space` and `decimal-format`. However, using II-paths, I see the highest improvement (17.93%) as inputs containing keyword (`attribute-set`) is also generated.

For *libxml2*, using inter paths, I do not see any improvements rather coverage is reduced as I waste 25% of the fuzzing time analyzing the paths. Inter paths can not find any rare inputs as it goes deep inside each and every procedure. Some of these procedures being analyzed for rare paths do not contain any complex program checks and due to exponential increase in the number of paths, it wastes time and cannot analyze significant procedures that contains complex program checks. Hence, the identified rare paths based on inter-paths are not actually rare paths for *libxml2* and guiding concolic execution using these rare paths does not generate rare inputs that can improve coverage performance.

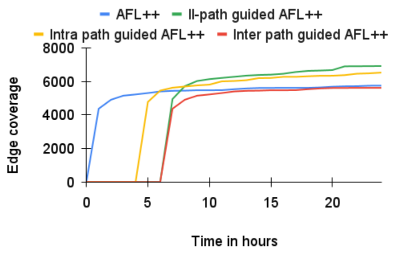
Identifying rare paths based on intra paths for *libxml2* can generate an input containing the



(a) cJSON



(b) libxslt



(c) libxml2

Figure 4.5: Coverage improvement comparison between different types of path-guided fuzzing. II-paths can generate more number of rare inputs compared to both intra and inter paths within a given amount of time and hence highest edge coverage is achieved by II-path guided fuzzing.

specific values: `DOCTYPE` and hence, I see coverage improvement. It can generate `DOCTYPE` as the branch conditions comparing to this specific value were inside the initial starting procedure. II-paths can generate inputs containing specific values: `DOCTYPE`, `ATTLIST`, `ENTITY` and `NOTATION`. These inputs help to achieve better coverage not only compared to AFL++ (20.35%) but also compared to both intra rare path (5.99%) and inter rare path (23.18%) analysis.

Chapter 5

Obtaining Information Leakage Bounds via Approximate Model Counting

Quantitative information flow (QIF) techniques measure information leakage in software systems [29,34,38,40,64] where the amount of information leaked is quantified using concepts such as channel capacity [31] and Shannon entropy [29, 34, 35, 38–40, 64]. In this chapter, I focus on symbolic quantitative information flow analysis based on symbolic execution.

Symbolic execution can be extended using model counting constraint solvers [45–47] to compute the probability of each execution path [19]. By tracking observable values for each execution path and computing the probability, it is possible to quantify the information leakage about program inputs using Shannon entropy [35,39]. However, since symbolic execution cannot explore all program paths in general, this approach cannot be used to provide sound bounds for the information leakage. Furthermore, model counting constraint solvers may not be able to provide a precise count for each constraint [48, 49], which can also lead to imprecision in analysis, and loss of soundness. In this chapter, I present techniques that, under both of these circumstances, still compute sound upper and lower bounds for the information leakage.

Below, I first provide an overview of the problem I address in this chapter (Section 5.1) and

```c
int compare(char *h, char *l) {
    int i = 0, res;
    while(h[i] == l[i]) {
        if(h[i] == '\0' && l[i] == '\0')
            break;
        i++;
    }
    res = h[i] - l[i];
    if(res == 0) return 0;
    else if(res < 0) return -1;
    else return 1;
}
```

Figure 5.1: Lexicographical comparison of strings in C

symbolic QIF analysis (Section 5.2). Then, I present my contributions which are: i) A framework that reduces finding sound bounds for information leakage in software to optimization problems (Section 5.3); ii) Solving the resulting optimization problems (Section 5.4); iii) Implementation of my approach (Section 5.5); and iv) Experimental evaluation of my approach (Section 5.6).

5.1 Overview

Consider the function for lexicographical string comparison in C (Fig. 5.1) with two inputs h and l. Let us assume that h is a secret input (i.e., a *high-security* input value), and l is a public input (i.e., a *low-security* input value). The comparison between h and l is done character by character. If all the characters of the input strings match, the function returns 0. But, if there is a mismatch, the function checks the mismatched characters to determine the lexicographical order and returns either -1 or 1 based on the order.

Note that the number of times the while loop body in the `compare` function is executed depends on the length of the matching prefix between h and l. If an attacker can differentiate execution paths by observing their execution time, they might be able to infer the length of the matching prefix between h and l, corresponding to a timing side-channel.

Using symbolic execution and a static cost model to track the observable values (for example, number of instructions executed for a path as an estimation of the execution time), I can automatically generate the path constraints with different costs and partition the execution paths based on the observable values. Let us call the disjunction of the path constraints with equivalent (i.e., indistinguishable) observation values the *observation constraint* for that observation. Observation constraints partition the domain of the secret input based on different observable values.

For simplicity, let us assume that h and l are strings of length 4, h is symbolic and value of l is the string `"info"` which results in 9 execution paths for the `compare` function. Based on the observable values, and assuming that two execution paths are distinguishable if they have observable values with a difference of at least 5 units, 9 path constraints are combined into 5 observation constraints.

In terms of quantifying information leakage, I can first note that there are 5 distinguishable observable values that partition the secret domain. Based on this, I can compute an upper bound for information leakage, called *channel capacity,* which is $\log_2 5 = 2.32$ bits. Channel capacity provides an upper bound for Shannon entropy and hence provides a sound upper bound for leakage.

By computing the probability of observing each observable value, and considering the partitioning of the input domain based on the observable values, it is possible to quantify the information leakage in terms of Shannon entropy.

Let ψ_i denote the observation constraint for the i^{th} observation, and c_i denote the model count (i.e., the number of satisfying solutions) for the observation constraint ψ_i. Assuming strings of length 4, the domain size (D) for the inputs is $256^4 = 4294967296$ (4 characters for h). Probability of the i^{th} observation is computed by dividing c_i by D. Table 5.1 shows path constraints (ϕ_j), model counts (c_i) and the observation probabilities (c_i/D) corresponding to the observation constraints (ψ_i) for the `compare` function. Using the probabilities, the information leakage can be calculated as 0.037 bits in terms of Shannon entropy.

There are two issues that can make this analysis approach unsound. First, since, symbolic

Table 5.1: Probabilistic symbolic execution results and observables for the `compare` function.

j	Path Constraint	i	Obs.	Model Count	Probability
1	$h[0] \neq \mathtt{i} \wedge h[0] - \mathtt{i} < 0$	1	57	4278190080	0.99609
2	$h[0] \neq \mathtt{i} \wedge h[0] - \mathtt{i} > 0$				
3	$h[0] = \mathtt{i} \wedge h[1] \neq \mathtt{n} \wedge h[1] - \mathtt{n} < 0$	2	83	16711680	0.00389
4	$h[0] = \mathtt{i} \wedge h[1] \neq \mathtt{n} \wedge h[1] - \mathtt{n} > 0$				
5	$h[0] = \mathtt{i} \wedge h[1] = \mathtt{n} \wedge h[2] \neq \mathtt{f} \wedge$ $h[2] - \mathtt{f} < 0$	3	109	65280	0.00001
6	$h[0] = \mathtt{i} \wedge h[1] = \mathtt{n} \wedge h[2] \neq \mathtt{f} \wedge$ $h[2] - \mathtt{f} > 0$				
7	$h[0] = \mathtt{i} \wedge h[1] = \mathtt{n} \wedge h[2] = \mathtt{f} \wedge$ $h[3] \neq \mathtt{o} \wedge h[3] - \mathtt{o} < 0$	4	135	255	5.94×10^{-8}
8	$h[0] = \mathtt{i} \wedge h[1] = \mathtt{n} \wedge h[2] = \mathtt{f} \wedge$ $h[3] \neq \mathtt{o} \wedge h[3] - \mathtt{o} > 0$				
9	$h[0] = \mathtt{i} \wedge h[1] = \mathtt{n} \wedge h[2] = \mathtt{f} \wedge$ $h[3] = \mathtt{o} \wedge h[3] - \mathtt{o} = 0$	5	127	1	2.32×10^{-10}

execution cannot guarantee coverage of all paths, the reported leakage may not be accurate. For example, for the `compare` function, assume that two of the paths, say paths 8 and 9, are not explored during symbolic execution, can I provide a sound bound for information leakage in such a scenario? Second, model counting constraint solvers cannot always provide a precise count. For example, assume that for observation 2, instead of reporting a count of 16711680, model counting constraint solver reports that the count is between 15192436 and 18382848. Can I provide a bound for information leakage in such a scenario?

For the `compare` function, Figure 5.2 and 5.3 show how entropy bound (red color representing upper bound and blue color representing lower bound) changes according to approximation tolerance of exact model count and percentage of input space unexplored, respectively. For an approximation tolerance ϵ, the given upper bound of the model count is guaranteed to be less than or equal to $(1+\epsilon)c_i$ and the given lower bound is guaranteed to be greater than or equal to $c_i/(1+\epsilon)$.

My main technical contribution in this chapter is to present a framework and techniques that compute sound upper and lower bounds for information leakage in both of these two scenarios and their combination (i.e., the information leakage is guaranteed to be within the bounds I compute).

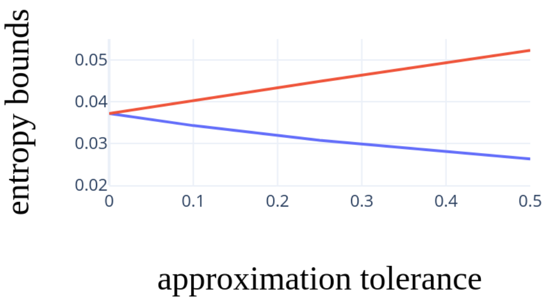


Figure 5.2: Entropy bounds vs. model count approximation

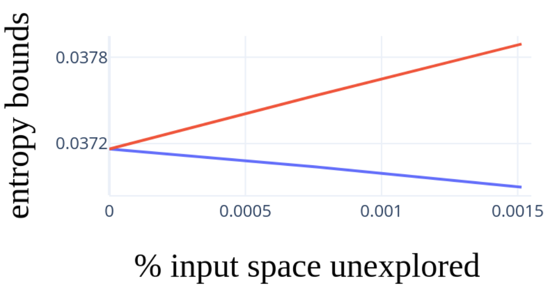


Figure 5.3: Entropy bounds vs. incomplete path coverage

5.2 Symbolic QIF Analysis

Symbolic Quantitative Information Flow (QIF) analysis has three components: 1) probabilistic symbolic execution via model counting, 2) extending symbolic execution via observable tracking, and 3) information theoretic analysis using Shannon entropy.

Algorithm 4 ProbabilisticSymbolicExecution(P)

```
1: $\Phi \leftarrow \emptyset$
2: $\Phi \leftarrow$ SymbolicExecution($P$)
3: $D \leftarrow \Phi$
4: for $\phi_i \in \Phi$ do
5:     $c_i \leftarrow$ ModelCount($\phi_i$)
6:     $p_i \leftarrow \frac{c_i}{D}$
7:     $p \leftarrow p \cup \{p_i\}$
8: return $p$
```

Probabilistic symbolic execution (Algorithm 1) is an extension of symbolic execution that computes probabilities of program paths [19]. Symbolically executing a program P using a symbolic execution engine I can collect a set of path conditions Φ where $\phi_i \in \Phi$ represents the i^{th} path condition.

In probabilistic symbolic execution, model counting constraint solvers are used to compute path probabilities. A model counting constraint solver computes the number of values (models) that satisfy a given constraint. Using a model counting constraint solver, I can compute the number of input values that satisfy a path condition. The number of satisfying inputs c_i for the i^{th} path condition ϕ_i is divided by the domain size of the inputs (D) to compute the execution path probability $p_i = \frac{c_i}{D}$.

Algorithm 5 ExtendedSymbolicExecution(P)

```
1: $\Psi \leftarrow \emptyset$
2: $(\Phi, \mathcal{O}, obs) \leftarrow$ SymbolicExecution($P$)
3: for $o \in \mathcal{O}$ do
4:     $\psi_o \leftarrow \bigvee_{\phi \in \Phi : obs(\phi) \sim o} \phi$
5:     $\Psi \leftarrow \Psi \cup \{\psi_o\}$
6: return $\Psi$
```

In order to quantify the information leakage, during symbolic execution, I need to keep track of either a main-channel observable or a side-channel observable for each path. For main-channels, I need to keep track of the concrete returned value from a program. For timing side-channels, I need to model the execution time of the function. This is typically done by keeping track of the number of instructions executed [39, 40, 95]. I assume that the observables are noiseless, i.e., multiple executions of the program with the same input value will result in the same observable value.

I extend symbolic execution (Algorithm 5) to return the set of possible observations $\mathcal{O}$, and a function *obs* that maps each path constraint $\phi \in \Phi$ to a corresponding observation $o \in \mathcal{O}$. Since it is not possible to extract information from program paths that have indistinguishable observable values, I combine observationally equivalent path constraints via disjunction (Algorithm 5, line 4), where o and o' are in the same equivalence class ($o \sim o'$) if and only if $|o - o'| < \delta$, i.e., the difference between them is below a given threshold. For example, path 1 and 2 in Table 5.1 are merged as both these paths have the same observable value (57), and, therefore, are indistinguishable. The resulting *observation constraints* (Ψ) characterize the information flow channel of the program.

With the set of observation constraints Ψ, I can transform Algorithm 4, by replacing the set of path constraints Φ by the set of observation constraints Ψ. Then, lines 4 to 8 compute the set of observation probabilities (rather than the path probabilities) where i^{th} observation probability is represented as $p(o_i)$. Table 5.1 shows model counts for the collected observation constraints and observation probabilities considering a domain size of 256^4 for the string compare example in Figure 5.1.

Information Entropy. I use Shannon entropy [29, 96] to measure the amount of information leaked about the secret input by program P from the observable values produced by P. Assuming a finite input domain, uniform distribution of program inputs, and a deterministic program, I define the Shannon entropy $\mathcal{H}$ of a program P as:

$$\mathcal{H}(P) = \sum_{o_i \in O} p(o_i) \log_2 \frac{1}{p(o_i)} \tag{5.1}$$

where $p(o_i)$ is the probability of observing o_i after executing P (from here on, all logarithms are assumed to be base 2), and I compute these probabilities using probabilistic symbolic execution with model counting as described above. Given observation constraints $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$, let c_i be the model count for ψ_i, I can plug $p(o_i) = c_i/D$ into Equation (5.1) to obtain

$$\mathcal{H}(P) = \sum_{i=1}^{n} \frac{c_i}{D} \log \frac{D}{c_i} \tag{5.2}$$

As I discuss in the next section, the entropy computed using this approach can be inaccurate (i.e., unsound) due to limitations of symbolic execution technique and model counting constraint solvers. My main contribution in this chapter is to provide techniques that compute sound bounds for information leakage by extending the basic symbolic QIF analysis approach outlined above.

Channel Capacity. A straightforward upper bound for Shannon entropy is the *channel capacity* [96]. For a program P, the worst case information leakage (i.e., the highest information leakage) occurs when all observations are equally likely: A uniform distribution. In this case

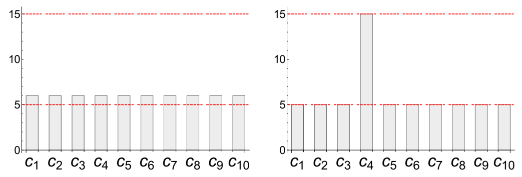


Figure 5.4: Model counts (c_i) for the 10 path constraints must sum to 60 and be between 5 and 15, upper and lower bounds as red dotted lines. Left: a solution for c_i that maximizes entropy. Right: a solution that minimizes entropy.

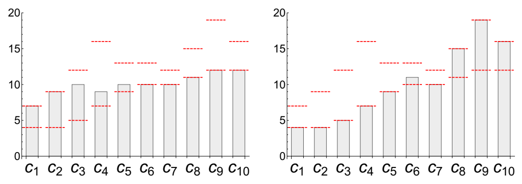


Figure 5.5: Model counts (c_i) for 10 path constraints must sum to 100. All have different upper and lower bounds (red dotted lines), sorted by lower bound. Left: a solution for c_i that maximizes entropy. Right: a solution that minimizes entropy.

Equation (5.2) simplifies to $\log|\Psi|$, where $|\Psi|$ is the number of observation constraints. Thus I define the channel capacity of a program P to be $CC(P) = \log|\Psi|$. By the channel capacity theorem I know that $\mathcal{H}(P) \leq CC(P)$.

5.3 Bounding Information Leakage

I now describe the analysis that take into account both reasons for *potential* unsoundness in measuring information leakage: (1) incomplete program path coverage and (2) approximate model counts. I begin with the simplest case in which coverage is complete and counts are exact, then move on to explain how to achieve sound bounds on information leakage (where the information leakage is guaranteed to be within the upper and lower bounds that I compute) in spite of the two challenges to soundness.

Full Path Coverage, Exact Counts

In this scenario, symbolic execution has performed an exhaustive exploration of the program paths and all model counts are exact. Hence, assuming deterministic program behavior and uniform distribution of program inputs, the probability of each observation is exactly known: $p_i = c_i/D$. In this case I apply Equation (5.1) to the probability distribution on observations induced by the model counts to compute the information leakage. This approach has been implemented and used in several related works [29, 30, 34, 39, 40].

Full Path Coverage, Approximate Counts

Suppose symbolic execution is able to explore all paths, but the model counts are not known precisely. For each model count c_i I know upper (u_i) and lower (l_i) bounds on the count. Shannon entropy is still given by Equation (5.1), but now each c_i varies over a constrained range. Instantiating each c_i by different values within the allowable range will result in a different leakage value. Therefore, I find the values of c_i that will maximize and minimize $\mathcal{H}$ by solving two constrained optimization problems:

$$\begin{array}{ll} \max\limits_{\{c_i\}} & \sum\limits_{i=1}^{n} \frac{c_i}{D} \log \frac{D}{c_i} \\ \text{s.t.} & l_i \leq c_i \leq u_i, \\ & \sum\limits_{i} c_i = D \end{array} \qquad\qquad \begin{array}{ll} \min\limits_{\{c_i\}} & \sum\limits_{i=1}^{n} \frac{c_i}{D} \log \frac{D}{c_i} \\ \text{s.t.} & l_i \leq c_i \leq u_i, \\ & \sum\limits_{i} c_i = D \end{array}$$

Techniques for solving these optimization problems are given in the next section. I first give two small examples.

Example 1. Consider a program with 10 path constraints and for each constraint my approximate model counter has determined that $5 \leq c_i \leq 15$. Assume domain size $D = 60$. I seek distributions that minimize and maximize the entropy. A well-known fact is that uniform distributions maximize entropy, so any uniform distribution resulting from counts between 5 and 15 will give an entropy of $\mathcal{H} = \log_2(10) = 3.322$ bits of information. One solution where $c_i = 6$ is shown in Figure 5.4 (left), but any uniform count for all path constraints would have also maximized the entropy. To minimize the entropy, I want to find a distribution that is, in some sense, as far from uniform as possible. In the example scenario, this happens when one of the counts is maximized and the rest are minimized, Figure 5.4 (right), giving an entropy of $\mathcal{H} = 3.189$ bits.

Example 2. Now suppose that my approximate model counter gives different upper and lower bounds for each path constraint (Figure 5.5, left). An entropy-maximizing distribution is one that gets as close to uniform as possible under the constraints, resulting in $\mathcal{H} = 3.307$. An entropy-minimizing distribution is one that is as non-uniform as possible (Figure 5.5, right) which occurs when all of the counts except one achieve either their minimum or maximum value within their bounds. In this case $\mathcal{H} = 3.140$.

Partial Path Coverage, Exact Counts

As is typical in symbolic execution, I place a bound on the exploration depth of the execution tree, and so some feasible program paths may not be explored. Yet, I would like to still provide sound upper and lower bounds on the expected information leakage. Here I cover this case when model counts are exactly known.

Suppose symbolic execution has provided observation constraints $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$ which constitute partial knowledge of the possible program observations. Given a constraint ψ_{in} on the allowable inputs of the program, I characterize the inputs that induce the remaining unexplored program behavior as $\psi_{rem} = \psi_{\text{in}} \wedge \neg \bigvee_{\phi_i \in \Psi} \phi_i$. I am then in a position to consider two extreme cases.

Case 1: Sound upper bound. Information leakage will be maximized when all possible program behaviors characterized by ψ_{rem} induce *unique and previously not discovered* observations, occurring when all remaining inputs result in different observations with respect to the already discovered observations, mapping directly to the sensitive input. Letting c_{rem} be the model count of ψ_{rem}, I adjust Equation (5.1) by introducing a second term to account for the entropy contribution from unexplored program behavior. A sound upper bound on information leakage is then given by

$$\sum_{i=1}^{n} \frac{c_i}{D} \log \frac{D}{c_i} + \frac{c_{\text{rem}}}{D} \log D$$

This scenario will result in the largest possible information gain because the conditional Shannon entropy of X given Y is maximized when X is uniformly distributed [96]. Thus, I assume the widest possible uniform distribution of undiscovered observations: there are c_{rem} possible undiscovered observations, all of which are equally likely. This gives the sound upper bound.

Case 2: Sound lower bound. An attacker gains the least amount of information when the yet-to-be-discovered program behaviors induce observations that are maximally redundant with respect to the already discovered observations: that is, all possible program behaviors characterized by ψ_{rem} induce indistinguishable observation to one of the previously discovered observation $o_{\max}$, where $o_{\max}$ is the observation that has the largest model count c_{max}. Given all the model counts c_i, I update the new counts c'_i as $c'_i = c_i$ for all i except $c'_{\max} = c_{\max} + c_{\text{rem}}$. A lower bound for information leakage is given by Equation (5.1) using c'_i in place of c_i.

To see why, consider Equation (5.1) adjusted as above where $c'_{max} = c_{\max} + c_{rem}$. It is straightforward to show that for any c_i other than $c_{\max}$, letting $c'_i = c_i + c_{rem}$ results in higher entropy. This claim follows from the fact that all $c_i \geq 1$, the monotonicty of the function $f(x) = x \log x$ when $x \geq 1$ (i.e. $f(x) < f(x + \delta)$ for $\delta > 0$), and rote manipulation of summations and logarithms. Note that, in this case, it is unnecessary to construct and count ψ_{in} since $c_{rem} = D - \sum_i c_i$. However, this construction will help us generalize to the case of incomplete path coverage and approximate counts.

Partial Path Coverage, Approximate Counts

Finally, I address the case in which symbolic execution does not perform a full exploration, and model counts are approximated by interval ranges. I use ψ_{rem} and c_{rem} as before. Again, there are two extreme cases to be analyzed that correspond to the sound upper and lower bounds on information leakage.

Case 1: Sound upper bound. Here I need to maximize over all possible feasible values of model counts c_i that are within the model counting lower and upper bounds, while also taking into consideration the number of observations corresponding to possible behaviors that have not been explored by symbolic execution. Each input with c_{rem} may induce one of the behaviors already observed, or it may induce new behavior. Therefore, I define new variables c_i', where c_i' is the number of inputs associated with c_{rem} that induce behavior i. Further, c_{rem} is approximate. Therefore, the following problem optimizes over all values of c_i, c_i', and c_{rem}. The first sum is for entropy contribution of observables already observed, the second sum is for the entropy contributions from new observations that could be induced from the inputs associated with c_{rem}, with a maximum of c_{rem} new observations.

$$\begin{aligned}
\max_{\{c_i, c_{\text{rem}}, c_i'\}} \quad & \sum_{i=1}^{n} \frac{c_i + c_i'}{D} \log \frac{D}{c_i + c_i'} + \sum_{i=n+1}^{n+c_{\text{rem}}} \frac{c_i'}{D} \log \frac{D}{c_i'} \\
\text{s.t.} \quad & l_i \le c_i \le u_i, \quad l_{\text{rem}} \le c_{\text{rem}} \le u_{\text{rem}}, \\
& \sum_i c_i + c_{\text{rem}} = D, \quad \sum_i c_i' = c_{\text{rem}}
\end{aligned}$$

Case 2: Sound lower bound. The optimization problem corresponding to minimizing the entropy is the the corresponding minimization problem to the first optimization problem from case 1.

$$\begin{aligned}
\min_{\{c_i, c_{\text{rem}}, c'_i\}} \quad & \sum_{i=1}^{n} \frac{c_i + c'_i}{D} \log \frac{D}{c_i + c'_i} + \sum_{i=n+1}^{n+c_{\text{rem}}} \frac{c'_i}{D} \log \frac{D}{c'_i} \\
\text{s.t.} \quad & l_i \le c_i \le u_i, \;\; l_{\text{rem}} \le c_{\text{rem}} \le u_{\text{rem}}, \\
& \sum_i c_i + c_{\text{rem}} = D, \;\; \sum_i c'_i = c_{\text{rem}}
\end{aligned}$$

5.4 Optimization for Bounding Leakage

I show how to solve the constrained optimizations problems for bounding information leakage using standard techniques and novel algorithms. I generally only present solutions to optimization problems corresponding to the case of approximate counts and full path coverage as I have shown that all other optimization problems reduce to those. I first apply the standard techniques of hill climbing and polyhedron vertex enumeration to solve the maximization and minimization problems respectively. Then, I present a greedy approach to the maximization problem and a branch and bound approach to the minimization problem. Finally, I present one set of algorithms that achieve non-tight bounds in $O(n)$ time in the case of unexplored paths and approximate counts.

Upper bound computation using Hill Climbing. Shannon entropy is a concave function with respect to event probabilities [97, 98]. There is only one local maximum for a concave function and hence, I use hill-climbing algorithm to find the maximum entropy value. The algorithm is guaranteed to find the maximum entropy when it converges. I start with a set of counts that satisfies the constraint $l_i \le c_i \le u_i$ and the constraint $\sum_{i=1}^{n} c_i = D$. Entropy is maximized when the probability distribution is balanced and this heuristic helps us to find the initial set of counts. Then, I repeatedly find the neighbors to the current set of counts and advance to the neighbor that yields the maximum entropy.

Convex polyhedron for lower bound computation Observe that the linear constraints on the counts, $l_i \leq c_i \leq u_i$ and $\sum_i c_i = D$, define a convex polyhedron. It has been demonstrated that the minimum of a concave function whose feasible region is a convex polyhedron always occurs at one of the extreme points of the polyhedron [99]. Hence, searching through the extreme points of the polyhedron enables us to find a sound lower bound for information leakage. I use Parma Polyhedra Library [100] to get the extreme points of the polyhedron defined by the above constraints and search through all the extreme points of the polyhedron to find the point that yields the lowest Shannon entropy, which provides the sound lower bound for information leakage.

I call the techniques to compute entropy bounds **HCP** (**H**ill climbing algorithm to compute upper bound and **C**onvex **P**olyhedron to compute lower bound of information leakage.

5.5 Implementation

I have implemented I techniques using symbolic execution tool KLEE [101], Automata-Based Model Counting constraint solver (ABC) [46], and an approximate model counter SearchMC [49].

ABC supports constraints in linear integer arithmetic and string theory. ABC computes exact model counts for linear integer arithmetic constraints and a subset of string constraints, and it provides upper bounds for combinations of string and integer constraints. By default, symbolic constraints generated in KLEE are in bit-vector theory. I adopted an extension of KLEE [61] which generates linear integer arithmetic constraints instead of bit-vector constraints. In this work, I use ABC only on linear integer arithmetic constraints generated using the integer extension of KLEE. Hence, for the programs that integer extension of KLEE can handle, when model count is computed using ABC, I get the exact model count, and the information leakage computed for the programs are exact if all the program paths are explored using symbolic execution. However, the integer extension of KLEE can not generate linear integer arithmetic constraints for all programs in general. For programs whose path constraints can only be represented using bit-vector theory, I use SearchMC to get an approximate count, a lower and an upper bound of the model count. SearchMC is an approximate model counter and it usually provides a lower and an upper bound of the log of the exact count. In specific cases, it provides an exact count. It can also provide probabilistically sound lower and upper bounds of the count if specific conditions are satisfied in the algorithm. I have integrated both ABC and SearchMC model counting tools in KLEE. Hence, I technique is capable of supporting all programs that can be analyzed by KLEE.

In addition to generating path constraints using KLEE, I track the number of bytecode instructions executed as observable in order to collect observation constraints.

Optimization techniques for sound symbolic QIF are implemented using a Python script which takes as input observation constraints collected using KLEE, and uses ABC and SearchMC for model counting. I use Parma Polyhedra Library [100] and its python interface PyParma [102] to get the extreme points of the polyhedron.

5.6 Experiments

Benchmarks I experimentally evaluate My techniques on 3 different sets of benchmarks collected from 3 earlier research works related to QIF analysis. The first set of benchmarks have been used in evaluating techniques to synthesize attacks for programs vulnerable to side-channels [43] (from Password Checker Insecure to Edit Distance in Table 5.2). The second set of benchmarks have been used to evaluate techniques to prove absence of timing side-channels [103]. The third set of benchmarks have been used for evaluating techniques to measure channel capacity [28]. This set of benchmarks consists of 8 programs representing information flow through the main channel (input to output) of the program. Note that, programs from the former two set of benchmarks are originally written in Java. For My evaluation, I convert all these programs

Table 5.2: Channel capacity and information leakage bound using exact model counts and approximation

Benchmarks	Secret (high) inputs	Secret Domain (bits)	Public (low) inputs	No. of Path Constraints	No. of Obs. Constraints	Channel Capacity	Leakage Exact Count	Leakage Approximate Count					
								$\epsilon = 0.05$		$\epsilon = 0.10$		$\epsilon = 0.25$	
								Lower Bound	Upper Bound	Lower Bound	Upper Bound	Lower Bound	Upper Bound
			`l = "JYgxYAHjiuJ3v3xi"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
Pass Check Insec	h	128	`l = "L34T6FLs4EUmEbQR"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
			`l = "9bViPfBFNZe799kx"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
			`l = "aj3n"`	16	1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Pass Check Sec	h	32	`l = "WO14"`	16	1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
			`l = "wH24"`	16	1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
			`l = "5dRb2Q8tDKehCfR6"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
String Equals	h	128	`l = "8PkRLio390Pur5Ey"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
			`l = "1SQKw52ZMXy8nsrh"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
			`l = 'a'`	2	2	1.000	0.960	0.945	0.970	0.932	0.980	0.887	0.998
Char Inequality	h	4	`l = 'm'`	2	2	1.000	0.986	0.974	0.993	0.962	0.998	0.928	1.000
			`l = 'z'`	2	2	1.000	0.999	0.995	1.000	0.987	1.000	0.960	1.000
			`l = "abcdefgh"`	17	17	4.087	0.995	0.980	1.009	0.964	1.021	0.917	1.043
String Char Inequality	h	64	`l = "mmmmmmmm"`	17	17	4.087	1.022	1.010	1.031	0.998	1.038	0.957	1.045
			`l = "wxyzwxyz"`	17	17	4.087	0.999	0.995	1.000	0.987	1.000	0.960	1.000
			`l = "g"`	9	9	3.170	0.291	0.279	0.303	0.268	0.315	0.241	0.351
IndexOf	h	64	`l = "5"`	9	9	3.170	0.291	0.279	0.303	0.268	0.315	0.241	0.351
			`l = "H"`	9	9	3.170	0.291	0.279	0.303	0.268	0.315	0.241	0.351
			`l = "ti9P"`	18	18	4.170	0.037	0.036	0.039	0.034	0.040	0.031	0.045
Compress	h	32	`l = "Z3lr"`	18	18	4.170	0.037	0.036	0.039	0.034	0.040	0.031	0.045
			`l = "ET4P"`	18	18	4.170	0.037	0.036	0.039	0.034	0.040	0.031	0.045
			`l = "znmd"`	497	412	8.687	0.588	-	0.613	-	0.637	-	0.710
Edit Distance	h	32	`l = "fee"`	187	150	7.229	0.441	-	0.460	-	0.478	-	0.533
			`l = "na"`	42	33	5.044	0.286	-	0.298	-	0.310	-	0.345
sanity_notaint_unsafe	sec	8	-	128	128	7.000	4.467	-	4.467	-	4.467	-	4.467
sanity_straightline_unsafe	a,b	64	-	3	2	1.000	0.811	0.792	0.830	0.773	0.849	0.722	0.896
sanity_unsafe	a,b	16	-	130	129	7.011	2.561	-	2.661	-	2.760	-	3.084
moresanity_arr_unsafe	taint	32	-	2	2	1.000	1.000	0.998	1.000	0.994	1.000	0.970	1.000
moresanity_lb_unsafe	taint	8	`a = 10`	3	3	1.585	1.313	1.296	1.326	1.276	1.337	1.220	1.364
			`a = 126`	3	3	1.585	1.313	1.296	1.326	1.276	1.337	1.220	1.364
			`a = -56`	3	3	1.585	1.313	1.296	1.326	1.276	1.337	1.220	1.364
	real_		`gs = "7CoFDeeua8ybpDCO"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
login_unsafe	pass-	128	`gs = "v8u7P3mKW2dxcJLj"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
	word		`gs = "D1gZ7GeWEPmp4Flv"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
			`p = "BrT9"`	21	17	4.087	0.147	0.141	0.153	0.135	0.159	0.121	0.177
timing_login_unsafe	u	32	`p = "ASfd"`	21	17	4.087	0.147	0.141	0.153	0.135	0.159	0.121	0.177
			`p = "pPKG"`	21	17	4.087	0.147	0.141	0.153	0.135	0.159	0.121	0.177
			`b = "8KWoIGaFRtUjGGtE"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
user_pwequal_unsafe	a	128	`b = "8Dver009XIBsk5zF"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
			`b = "1VDDySiw1Cb001qw"`	17	17	4.087	0.037	0.036	0.039	0.034	0.040	0.031	0.045
copy	h	8	-	256	256	8.000	8.000	-	8.000	-	8.000	-	8.000
implicit	h	8	-	8	8	3.000	0.221	0.000	0.221	0.000	0.221	0.000	0.221

to C as My current implementation supports programs written in C only. I implement necessary functions from different Java classes in C. These modifications neither provide additional benefit to the proposed technique nor affect the evaluation results. I select these benchmarks as I believe that these programs represent a wide variety of scenarios for QIF analysis. My evaluation results for the 3 sets of benchmarks are shown in Tables 5.2 and 5.3.

Experimental Setup For all the experiments, I use a desktop machine with an Intel Core i5-2400S 2.50 GHz CPU and 32 GB of DDR3 RAM running Ubuntu 18.04 LTS, with a Linux

Table 5.3: Channel capacity and sound information leakage bound using an approximate model counter

Bench-marks	Secret Inputs	Secret Domain (bits)	Public Inputs	No. of Path Cons.	No. of Obs. Cons.	Channel Capa-city	Information Leakage	
							Lower Bound	Upper Bound
modPow1_unsafe	expo-nent	16	base = 4378, mod = 10000	18	18	4.170	1.855	2.119
			base = 43780, mod = 50000	18	18	4.170	1.848	2.127
			base = 8796, mod = 328765	18	18	4.170	1.856	2.198
modPow2_unsafe	expo-nent	16	base = 54, mod = 4000	18	18	4.170	1.855	2.122
			base = 4378, mod = 10000	18	18	4.170	1.859	2.117
			base = 287, mod = 6098	18	18	4.170	1.861	2.114
gpt14_unsafe	b	10	a = 54, c = 4000	1024	236	7.883	7.509	7.509
			a = 34, c = 100	1024	124	6.954	6.394	6.394
			a = 2345, c = 78350	1024	63	5.977	4.854	4.854
k96_unsafe	x	12	y = 5836, n = 3268	4096	51	5.672	4.342	4.342
			y = 6736928, n = 1073741824	4096	51	5.672	4.342	4.342
			y = 47, n = 3789	4096	51	5.672	4.342	4.342
masked_copy	h	32	-	16	16	4.000	4.000	4.000
checked_copy	h	32	-	17	16	4.000	1.1E-7	1.1E-7
pop_count	h	32	-	33	33	5.044	-	3.656
mix_copy	h	8	-	256	256	8.000	8.000	8.000
div_2	h	8	-	128	128	7.000	7.000	7.000
mul_2	h	8	-	256	256	8.000	8.000	8.000

4.15.0-20 64-bit kernel.

Experimental Results I now discuss My experimental results from the perspectives of bounding the information leakage given approximation of model count and incomplete path coverage. I compare the efficiency and precision of multiple algorithms in computing the lower and upper bound of the information leakage.

First, I experimentally compare the precision of channel capacity and Shannon entropy in quantifying information leakage. Next, I show results for lower and upper bound using the Shannon entropy based measure and considering full path coverage and approximation of model count. I also show results to compare the time required by different algorithms to compute the entropy bounds. Then, I show results from different perspectives (entropy bound, model counting time, entropy bound computation time) considering both approximation of model counting and incomplete path coverage (percentage of input space unexplored).

Channel capacity vs Shannon entropy

Channel capacity makes worst case assumptions about the probability distributions of the inputs. One of the recent works [31] argues channel capacity as the measure for precise computation of information leakage and provides an automatic and scalable manner for computing it. My experimental results show that Shannon entropy provides a more precise assessment of information leakage for analyzing quantitative information flow of a program compared to channel capacity.

For majority of the programs across 3 benchmarks as show in Tables 5.2 and 5.3), Shannon entropy based information leakage provides a significantly tighter bound compared to the channel capacity even when approximate counts are used. Out of 28 programs in total, the upper bound for information leakage computed using Shannon entropy is less than or a tighter bound compared to the channel capacity for 20 programs. For 10 out of these 28 programs, Shannon entropy based leakage is an order of magnitude lower than the channel capacity. Channel capacity and Shannon entropy computes same bits of information leakage for some programs in the benchmarks. This is because all the observations equally partition the input domain. For examples where Shannon entropy based leakage is order of magnitude lower than channel capacity is due to the fact that channel capacity considers only the number of observations whereas actually what happens is among all the observations one of the particular observation covers most of the input cases and hence distributes the domain in a very imbalanced manner. Though, channel capacity identifies these programs as very leaky, in reality, these programs are leaking very small amount of information. This demonstrates that, using channel capacity some programs could be marked as vulnerable to information leakage based on a given threshold, even though the leakage may be below the threshold if one computes the leakage in terms of

Shannon entropy.

Computing channel capacity is easier compared to Shannon entropy as it does not need to compute individual observation probabilities, rather it is sufficient to compute the number of observations and report the logarithm of the number of observations. On the other side, Shannon entropy based analysis provides expected information gain about the input and needs to compute each and every observation probabilities. Hence, Shannon entropy measure for information leakage computation is computationally expensive compared to channel capacity.

Leakage computation and low inputs

In addition to high (secret) inputs, programs can also have low (public) inputs. I now discuss different scenarios with respect to low inputs in the program (as demonstrated by the programs I use in My experiments):

No low inputs: Program does not have any low inputs and leakage depends only on the high inputs. For example, *sanity_notaint_unsafe* in Table 5.2 and *masked_copy* in Table 5.3.

Leakage is independent of low inputs: Program has low inputs but observation probability distribution does not depend on the low inputs. As a result, same amount of information is leaked for any low input provided to the program. For example, *Pass Check Insecure* in Table 5.2 and *k96_unsafe* in Table 5.3.

Leakage depends on low inputs: Program has low inputs and observation probability distribution is different for different low inputs. Hence, depending on the low inputs provided to the program, different amount of information is leaked. For example, *Char Inequality* in Table 5.2 and *modpow1_unsafe* in Table 5.3. Looking into the *Char Inequality* program, I see that setting the high input as symbolic and symbolically executing the program for different low inputs, two observation constraints are generated. But, for different low inputs, the constraints are different. For example, setting l (low input) as character `'a'`, two observation constraints are $h > \texttt{a}$ and $h \leq \texttt{a}$. For a domain size of 256 for h, model count for the two constraints are 98 and 158 respectively and hence observation probability distribution is [0.38, 0.62] and hence the amount of leakage is 0.960. Similarly, for low input `'m'`, probability distribution is [0.43, 0.57] and amount of leakage is 0.986 and for low input `'z'`, probability distribution is [0.48, 0.52] and leakage is 0.999.

Finding a low input value that maximizes information leakage is called *attack synthesis*. It is possible to combine attack synthesis techniques with the techniques I present in this chapter to find the low inputs. By maximizing and minimizing the leakage lower and and upper bounds of leakage can be obtained for low dependent programs. Attack synthesis based on parameterized model counting and numerical optimization has been developed in [104]. I have developed an attack synthesis technique based on metaheuristic search [43] which I will discuss in detail in chapter 6.

Table 5.4: Comparison between channel capacity and sound information leakage bounds using HCP for login_unsafe example from Blazer benchmarks

unexp. input space (%)	ϵ	HCP Lower Bound	HCP Upper Bound	Channel Capacity
9.09×10^{-11}	0.05	0.036	0.039	123.608
	0.10	0.034	0.040	124.541
	0.25	0.031	0.045	125.678
5.96×10^{-6}	0.05	0.036	0.039	123.608
	0.10	0.034	0.040	124.541
	0.25	0.031	0.045	125.678
3.91×10^{-1}	0.05	0.000	0.531	123.716
	0.10	0.000	0.556	124.596
	0.25	0.000	0.632	125.700

Full Path Coverage, Approximate Count

Here, I provide results to show the changes in lower and upper bound of the information leakage based on different approximation tolerance (ϵ) values. In Table 5.2, I show exact information leakage given exact model counts that I get using ABC [46]. Then, given an exact count (c) of an observation constraint, I use different values of approximation tolerance (ϵ) to generate approximate counts using formulas $(1+\epsilon)c_i$ and $c_i/(1+\epsilon)$ and compute upper and lower bounds for information leakage using these approximate counts. As I expect, I find that increasing the value of ϵ decreases the lower bound I compute for information leakage and increases the upper bound I compute for information leakage, overall computing looser bounds of the exact information leakage. For all programs, the computed information leakage bounds contain the exact information leakage as expected.

For 10 out of the 28 programs, it is not possible to compute exact information leakage using exact model counting since ABC can not handle non-linear constraints generated from these programs. I use the approximate model counter SearchMC [49] for these programs and show results in Table 5.3. Note that, lower and upper bounds of the model count computed by SearchMC are probabilistically sound. I use confidence level of 0.95 and threshold value of 0.5 for SearchMC to collect the counts. I ran each program 5 times using SearchMC and report the average value.

Incomplete path coverage, approximate count

Till now, I have shown results for computing sound bounds of information leakage given lower and upper bounds of the model count considering complete path coverage by symbolic execution. I now present and discuss results for computing sound lower and upper bounds of information leakage considering path coverage is incomplete and hence all of the input space is not explored. From the results, I find that as percentage of unexplored input space increases, leakage bounds get looser and looser, similar to case with the increasing approximation tolerance (ϵ). This pattern of result (increase in the range of entropy bound depending on both model count approximation and incomplete path coverage) exists across all programs as I expect. Below I discuss the results for the login_unsafe example from the Blazer benchmark.

In Table 5.4, I also present entropy bounding comparison between HCP and channel capacity. The results clearly indicate that the information leakage bounds I compute is orders of magnitude lower compared to information leakage computed by channel capacity even though the complete input space is not explored and the model count is an approximation of the exact model count.

Overall, experimental results show that the techniques I developed can deal with different scenarios arising for sound analysis of quantitative information flow and these techniques can be used to trade off between tightness of information leakage bound.

Chapter 6

Side-Channel Attack Synthesis

In this chapter, I focus on the problem of assessing side-channel vulnerabilities in software in a quantitative manner. By synthesizing an attack based on quantification of information leakage, I provide an exploit demonstrating the side-channel vulnerability of the function. The synthesized attack consists of a sequence of public inputs that a malicious user can use in an adaptive manner to leak information completely or partially about a secret in the program by observing side-channel behavior.

Given a function that performs computation over secret values unknown to users, I synthesize a side-channel attack against that function. Failure of the approach to find an attack increases the confidence that the function is safe against side-channel attacks. The approach uses symbolic execution to extract constraints that characterize the relationship between the secret values in the program, attacker controlled inputs, and side-channel observations.

My technique can automatically synthesize side-channel attacks for programs that manipulate both unbounded string and numeric values [43]. I demonstrate an approach that maximizes the information gain about the secret in each attack step and use meta-heuristics for searching the input space during attack synthesis, resulting in a generalized attack synthesis approach. To improve efficiency of the attack synthesis technique, I also present an incremental attack synthesis approach [44] based on incremental automata-based model counting that reuses the results from prior attack steps in order to improve the efficiency of attack synthesis. I implement the attack synthesis approach for Java programs using symbolic execution tool SPF and ABC model counter and present experiments demonstrating realistic attack scenarios.

My approach consists of following two phases:

(1) Static Analysis Phase: In this phase, I perform symbolic execution on a program marking the secret input and the attacker controlled input as symbolic. I keep track of a side-channel observation for each path of the program, assuming that the observable values are noiseless. Augmenting symbolic execution to a return function, I map a path constraint to an observation. Then, I merge the path constraints with indistinguishable observation values via disjunction to collect the observation constraints.

(1) Attack Synthesis Phase: In this phase, I fix a value for the secret, unknown to the attacker, which I will try to reveal completely or partially at the end of the attack. I maintain a constraint on the secret value which is initially *TRUE*, representing it can be anything from a defined finite domain. The constraint is then iteratively updated based on observation constraints and maximization of information gain at each attack step. To compute information gain Shannon entropy formula from information theory is used. I investigate and compare several methods for selecting the input at each step based on meta-heuristics for maximizing the amount of information gained and automata-based techniques for constraint solving and model counting. My attack synthesis approach is adaptive, taking into account the information learned about the secret in previous steps while choosing the attacker controlled input for the next attack step; and it is incremental, re-using the results from prior iterations in order to improve the performance of each attack synthesis step. Moreover, my attack synthesis approach can handle unbounded string constraints in addition to linear arithmetic constraints.

After a motivating example, I will explain phases of my analysis in order, followed with experimental evaluations and related work on quantification of information leakage, side-channel

```
public Boolean checkPIN(String h, String l){
  for (int i = 0; i < 4; i++)
    if (h.charAt(i) != l.charAt(i))
      return false;
  return true;
}
```

Figure 6.1: PIN checking example.

analysis and automated synthesis of attacks.

6.1 Motivation

Motivating Example 1. Consider a PIN-based authentication function (Fig. 6.1) with inputs: 1) a secret PIN h, and 2) a user input, l. Both h and l are strings of digit characters ("0"–"9") of length 4. I have adopted the nomenclature used in security literature where h denotes the *high-security* value (the secret PIN) and l denotes the *low-security* input value, (the input that the function compares with the PIN). The function compares the PIN and the user input character by character and returns `false` as soon as it finds a mismatch. Otherwise it returns `true`.

This function has a timing side-channel and one can infer information about the secret h by measuring the execution time. In this implementation of `checkPIN` each length of the common prefix of h and l, the number of bytecode instructions that will be executed will differ which may cause an observable difference in execution time. Notice that if h and l have no common prefix, then `checkPIN` will have the shortest execution since the loop body will be executed only once; this corresponds to execution of 63 Java bytecode instructions. If h and l have a common prefix of one character, I see a longer execution since the loop body executes twice (78 instructions). In the case that h and l match completely, `checkPIN` has the longest execution (123 instructions).

If I assume that differences in the number of bytecode instructions are observable by measuring the execution time, then there are 5 observable values since there are 5 execution paths with different lengths, proportional to the length of the common prefix of h and l. In general, using the number of executed bytecode instructions as a measurable observation can result in observations that are indistinguishable in practice. Thus, I combine observations into indistinguishability intervals $o \pm \delta$ using an observability threshold δ. For this example assume that differences among execution path lengths are above this threshold.

Given this side-channel, an attacker can choose an input and use the timing observation to determine how much of a prefix of the input has matched the secret. In order to automate this process, My approach starts with automatically generating the path constraints and corresponding execution costs (in terms of number of executed bytecode instructions) using symbolic execution (Table 6.1). It merges path constraints based on the observability threshold, resulting in a set of observability constraints. It then uses these constraints to synthesize an attack which determines the value of the secret PIN. I make use of an uncertainty function, based on Shannon entropy, to measure the progress of an attack (Section 6.3). Intuitively, the attacker's uncertainty, $\mathcal{H}$ starts off at some positive value corresponding to the initial uncertainty of the sercret, and decreases during the attack. When $\mathcal{H} = 0$, the attacker has fully learned the secret (Table 6.2).

Table 6.1: Observation constraints generated by symbolic execution of the function in Figure 6.1.

i	Observation Constraint, ψ_i	o
1	$charat(l,0) \neq charat(h,0)$	63
2	$charat(l,0) = charat(h,0) \wedge$ $charat(l,1) \neq charat(h,1)$	78
3	$charat(l,0) = charat(h,0) \wedge$ $charat(l,1) = charat(h,1) \wedge$ $charat(l,2) \neq charat(h,2)$	93
4	$charat(l,0) = charat(h,0) \wedge$ $charat(l,1) = charat(h,1) \wedge$ $charat(l,2) = charat(h,2) \wedge$ $charat(l,3) \neq charat(h,3)$	108
5	$charat(l,0) = charat(h,0) \wedge$ $charat(l,1) = charat(h,1) \wedge$ $charat(l,2) = charat(h,2) \wedge$ $charat(l,3) = charat(h,3)$	123

Table 6.2: Attack inputs (l), uncertainty about the secret ($\mathcal{H}$), and observations (o). Prefix matches are shown in **bold**.

Step	$\mathcal{H}$	l	o	Step	$\mathcal{H}$	l	o
1	13.13	“8299”	63	15	5.906	“**13**92”	93
2	12.96	“0002”	63	16	5.643	“**13**16”	93
3	9.813	“**1**058”	78	17	5.321	“**13**08”	93
4	9.643	“**1**477”	78	18	4.906	“**13**62”	93
5	9.451	“**1**583”	78	19	4.321	“**13**78”	93
6	9.228	“**1**164”	78	20	3.169	“**133**8”	108
7	8.965	“**1**950”	78	21	3.000	“**133**2”	108
8	8.643	“**1**220”	78	22	2.807	“**133**4”	108
9	8.228	“**1**786”	78	23	2.584	“**133**3”	108
10	7.643	“**1**817”	78	24	2.321	“**133**0”	108
11	6.643	“**1**664”	78	25	2.000	“**133**5”	108
12	6.491	“**13**42”	93	26	1.584	“**133**6”	108
13	6.321	“**13**28”	93	27	0.000	“**1337**”	123
14	6.129	“**13**86”	93				

Suppose that the secret is “1337”. The initial uncertainty is $\log_2 10^4 = 13.13$ bits of information (assuming uniform distribution). My attack synthesizer generates input “8229” at the first step and makes an observation with cost 63, which corresponds to ψ_1. This indicates that $charat(h,0) \neq 8$. Similarly, a second synthesized input, “0002”, implies $charat(h,0) \neq 0$ and the uncertainty is again reduced. At the third step the synthesized input “1058” yields an observation of cost 78. Hence, ψ_2 is the correct path constraint to update My constraints on h, which becomes

$$charat(h,0) \neq 8 \wedge charat(h,0) \neq 0 \wedge charat(h,0) = 1 \wedge charat(h,1) \neq 0$$

I continue synthesizing inputs and updating the constraints on h, which tell us more information about h, until the secret is known after 27 steps. At the final step, I make an observation which corresponds to ψ_5 indicating a full match and the remaining uncertainty is 0. As in this example, the goal of My search for attack inputs is to drive the entropy that characterizes the remaining uncertainty about the secret to 0. Thus, I propose entropy optimization techniques.

This particular type of attack is called a *segment attack* which is known to be a serious sMy ce of security flaws [39, 105–108], and it is exponentially shorter than a brute-force attack. My approach automatically synthesizes a segment attack.

Motivating Example 2. Consider another example (Figure 6.2). If secret value h is lexicographically smaller than user input l, the execution time of *stringInequality* corresponds to 47 instructions, and 62 otherwise. Symbolically executing the *stringInequality* method (note that, I do not symbolically execute the *compareTo* method from Java's string library but capture it as a string constraint directly), two path constraints are inferred with distinguishable observations shown in Table 6.3. For simplicity, consider the secret domain to be from "AA" to "ZZ" ($26^2 = 676$ strings), the secret value is "LL" and the first attack input is "AA". In Table 6.4 I show an attack that recovers the secret in 20 attack steps.

I can generate an attack like the one shown in Table 6.4 by finding a satisfying solution (i.e., model) to the constraints on the low variable that is consistent with the observations about the secret I have accumulated so far. I call this the Model-based (M) approach (see section 6.4), and this approach does generate optimal segment attacks as I discussed above for the example shown in Fig. 6.1. However, for the example shown in Figure 6.2 the Model-based approach cannot generate an optimal attack.

The attack shown in Table 6.4 recovers the secret but it is not optimal in terms of the length of the attack. In order to generate an optimal attack I have to choose an input that maximizes the amount of information leaked in each attack step. Then, I can generate the attack shown in Table 6.5 which is optimal and requires only 9 steps. This corresponds to a binary search, finding the middle point to divide the domain of secret value in a balanced way. For My example, the domain size d is 26^2 and taking $\log_2 d$, I get $\lceil 9.40 \rceil = 10$ attack steps in the worst case. In order to generate the optimal attack automatically, I need to construct an objective function (see section 6.3) characterizing the information gain for each attack step and use optimization

```java
public static void stringInequality(String h, String l) {
    if(h.compareTo(l) <= 0) {
      for (int i = 1; i > 0 ; i--);
    } else {
      for (int i = 5; i > 0 ; i--);
    }
}
```

Figure 6.2: String inequality example.

techniques (see section 6.4) to maximize the objective function.

Let us have a look at the constraints on secret value h at each attack step for the optimal attack from Table 6.6. At each attack step I gain new information about the secret value h and a new constraint is added to the already existing constraint C_h. The constraint C_h grows and becomes more complex in each attack step. Constraint solving and model counting are the most expensive parts of My approach. So, if I can reuse prior solutions to constraint solving and model counting to take advantage of the incremental nature of attack synthesis, I can increase the efficiency of My approach. I call this approach *incremental attack synthesis* (see section 6.3) and demonstrate that it improves the efficiency of attack synthesis significantly (see section 6.5).

Table 6.3: Observation constraints of the function in Figure 6.2

i	ψ_i	o
1	$h \leq l$	42
2	$h > l$	67

Table 6.4: Non-optimal attack

Step	$\mathcal{H}$	l	o	Step	$\mathcal{H}$	l	o
1	9.40	“AC"	67	11	7.56	“PJ"	42
2	9.39	“AE"	67	12	6.82	“PI"	42
3	9.39	“JZ"	67	13	6.80	“NA"	42
4	8.70	“XE"	42	14	5.70	“LZ"	42
5	8.41	“XB"	42	15	4.64	“LI"	67
6	8.40	“KQ"	67	16	4.00	“LR"	42
7	8.33	“XA"	42	17	3.00	“LK"	67
8	8.32	“KU"	67	18	2.58	“LO"	42
9	8.30	“SI"	42	19	1.58	“LM"	42
10	7.60	“KZ"	67	20	0.00	“LL"	42

Table 6.5: Optimal attack

Step	$\mathcal{H}$	l	o
1	8.40	“MZ"	42
2	7.40	“GM"	67
3	6.40	“JS"	67
4	5.43	“LI"	67
5	4.39	“MD"	42
6	3.32	“LS"	67
7	2.32	“LN"	67
8	1.00	“LK"	67
9	0.00	“LL"	42

Table 6.6: Incremental nature of constraints at each step of adaptive attack.

Attack step	Attack input	Constraint on secret value, C_h
1	"MZ"	$h <=$ "MZ"
2	"GM"	$h <=$ "MZ" $\wedge h >$ "GM"
3	"JS"	$h <=$ "MZ" $\wedge h >$ "GM" $\wedge h >$ "JS"
...	...	...
9	"LL"	$h <=$ "MZ" $\wedge h >$ "GM" $\wedge h >$ "JS"$\wedge$ $h >$ "LI" $\wedge h <=$ "MD" $\wedge h <=$ "LS"$\wedge$ $h >$ "LN" $\wedge h >$ "LK" $\wedge h <=$ "LL"

6.2 Synthesizing Adaptive Attacks

I use a two-phase attack synthesis approach as shown in Fig. 6.3 and Algorithm 6. I consider a function F that takes as input a secret $h \in \mathbb{H}$ and an attacker-controlled input $l \in \mathbb{L}$ and that generates side-channel observations $o \in \mathbb{O}$.

Static Analysis Phase. In the first phase I generate observation constraints from F as shown in Algorithm 7. First, I perform symbolic execution on F with the secret (h) and the attacker controlled input (l) marked as symbolic [109,110]. Symbolic execution runs F on symbolic rather than concrete inputs resulting in a set of path constraints Φ. Each $\phi \in \Phi$ is a logical formula that characterizes the set of inputs that execute some path in F. During symbolic execution, I keep track of a side-channel observation for each path. For timing side-channels, as in other works in this area, I model the execution time of the function by the number of instructions executed [39,95,104]. I assume that the observable values are noiseless, i.e., multiple executions of the program with the same input value will result in the same observable value. I augmented symbolic execution to return a function that maps a path constraint ϕ to an observation o. Since an attacker cannot extract information from program paths that have indistinguishable side-channel observations, I combine observationally similar path constraints via disjunction (Algorithm 7, line 4), where I say that o and o' are in the same equivalence class ($o \sim o'$) if and only if $|o - o'| < \delta$. The resulting *observation constraints* (denoted ψ_o and Ψ) characterize

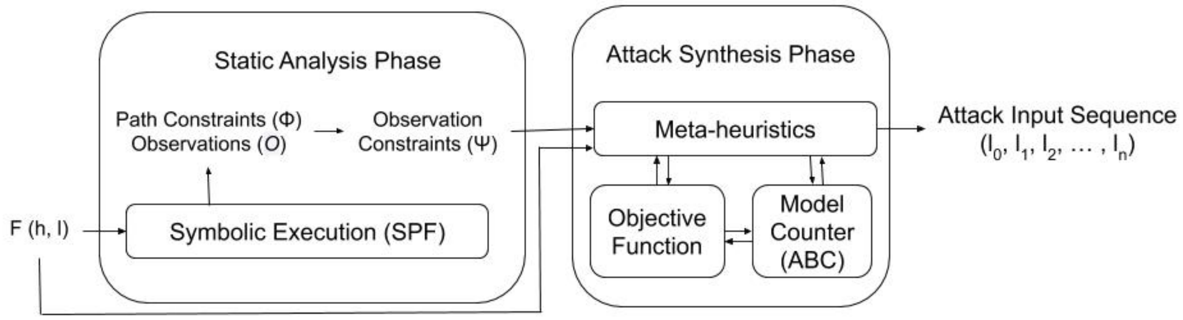


Figure 6.3: Overview of Attack Synthesis Approach

the relationship between the secret (h) the attacker input (l) and indistinguishable side-channel observations (o).

Algorithm 6 SYNTHESIZEATTACK($F(h,l), C_h, h^*$)
This algorithm calls the GENERATECONSTRAINTS and RUNATTACK functions to synthesize adaptive attacks.

```
1: $\Psi \leftarrow$ GENERATECONSTRAINTS($F(h,l)$)
2: RUNATTACK($F(h,l), \Psi, C_h, h^*$)
```

Attack Synthesis Phase. The second phase synthesizes a sequence of inputs that allow an attacker to adaptively learn the secret (Algorithm 8). During this phase, I fix a secret h^*, unknown to the attacker. I maintain a constraint C_h on the possible values of the secret h. Initially, C_h merely specifies the domain of the secret. I call algorithm ATTACKINPUT-SA, which uses the simulated annealing technique to maximize information gain about the secret expressed as entropy (as discussed in section 6.4), to determine the input value l^* for the current attack step. Then, the observation o that corresponds to running the program under attack with h^* and l^* is revealed by running the function using the public input l^*. I update C_h to reflect the new constraint on h implied by the attack input and observation—I instantiate the corresponding observation constraint, $\psi_o[l \mapsto l^*]$, and conjoin it with the current C_h (line 5). Based on C_h, I compute an uncertainty measure for h at every step using Shannon entropy [96,111], denoted $\mathcal{H}$ (Section 6.3). The goal is to generate inputs which drive $\mathcal{H}$ as close as possible to zero, in which case there is no uncertainty left about the secret and the secret is fully known. This attack synthesis phase is repeated until it is not possible to reduce the uncertainty, $\mathcal{H}$, any further.

Algorithm 7 GenerateConstraints($F(h, l)$)
Performs symbolic execution on function F with secret string h and attacker-controlled string l. The resulting path constraints are combined according to indistinguishability of observations.

```
1: $\Psi \leftarrow \emptyset$
2: $(\Phi, \mathcal{O}, obs) \leftarrow$ SymbolicExecution($F(h, l)$)
3: for $o \in \mathcal{O}$ do
4:     $\psi_o \leftarrow \bigvee_{\phi \in \Phi : obs(\phi) \sim o} \phi$
5:     $\Psi \leftarrow \Psi \cup \{\psi_o\}$
6: return $\Psi$
```

Algorithm 8 RunAttack($F(h, l), \Psi, C_h, h^*$)
Synthesizes a sequence of attack inputs, l^*, for $F(h, l)$, given observation constraints Ψ, initial constraints on h (C_h), and unknown secret h^*.

```
1: $\mathcal{H} \leftarrow$ Entropy($C_h$)
2: while $\mathcal{H} > 0$ do
3:     $l^* \leftarrow$ AttackInput-SA($C_h, \Psi$)
4:     $o \leftarrow F(h^*, l^*)$
5:     $C_h \leftarrow C_h \wedge \psi_o[l \mapsto l^*]$
6:     $\mathcal{H} \leftarrow$ Entropy($C_h$)
```

6.3 Incremental Attack Synthesis

In this section, I first describe the objective function I use to guide the synthesis of each attack step. Then, I discuss the use of automata-based model counting for computing the objective function. Finally, I describe My incremental approach to attack synthesis that reuses results of model counting queries from prior steps for improving efficiency.

Objective Function for Information Gain

Here I derive an objective function to measure the amount of information an attacker expects to gain by choosing an input value l_{val} to be used in the attack search heuristics discussed in section 6.4. In the following discussion, H, L, and O are random variables representing high-security input, low-security input, and side-channel observation, respectively. I use entropy-based metrics from the theory of quantitative information flow [29]. Given probability function $p(h)$, the *information entropy* of H, denoted $\mathcal{H}(H)$, which I interpret as the initial *uncertainty* about the secret, is

$$\mathcal{H}(H) = -\sum_{h \in \mathbb{H}} p(h) \log_2 p(h) \tag{6.1}$$

Given conditional distributions $p(h|o,l)$, and $p(o|l)$, I quantify the attacker's expected *updated uncertainty* about h, given a candidate choice of $L = l_{val}$, with the expectation taken over all possible observations, $o \in O$. I compute the *conditional entropy of* H *given* O *with* $L = l_{val}$ as

$$\mathcal{H}(H|O, L = l_{val}) = -\sum_{o \in \mathbb{O}} p(o|l_{val}) \sum_{h \in \mathbb{H}} p(h|o, l_{val}) \log_2 p(h|o, l_{val}) \tag{6.2}$$

Now I can compute the expected amount of information *gained* about h by observing o after running the function F with a specific input l_{val}. The *mutual information* between H and O, given $L = l_{val}$ denoted $\mathcal{I}(H;O|L = l_{val})$ is the difference between the initial entropy of H and the conditional entropy of H given O when $L = l_{val}$:

$$\mathcal{I}(H;O|L = l_{val}) = \mathcal{H}(H) - \mathcal{H}(H|O, L = l_{val}) \tag{6.3}$$

Equation (6.3) serves as My objective function. Providing input $l_{val} = l^*$ which maximizes $\mathcal{I}(H;O|L = l_{val})$ maximizes information gained about h. Equations (6.1) and (6.2) rely on $p(h)$, $p(o|l)$, and $p(h|o,l)$, which may change at every step of the attack. Recall that during the attack, I maintain a constraint on the secret, C_h. Assuming that all secrets that are consistent with C_h are equally likely, at each step, I can compute the required probabilities using model counting. Given a formula f, performing model counting on f gives the number of satisfying solutions for F, which I denote $\#f$. Thus, I observe that $p(h) = 1/\#C_h$ if h satisfies C_h and is 0 otherwise. Hence, Equation 6.1 reduces to $\mathcal{H}(H) = \log_2(\#C_h)$.

Algorithm 7 gives us side-channel observations $\mathcal{O} = \{o_1, \ldots, o_n\}$ and constraints over h and l corresponding to each o_i, $\Psi = \{\psi_1, \ldots, \psi_n\}$. The probability that the secret has a particular value, constrained by a particular ψ_i, for a given l_{val} can be computed by instantiating ψ_i with l_{val} and then model counting. Thus, $p(h|o_i, l_{val}) = 1/\#(C_h \wedge \psi_i)[l \mapsto l_{val}]$. Similarly, $p(o_i|l_{val}) = \#(C_h \wedge \psi_i)[l \mapsto l_{val}]/\#C_h[l \mapsto l_{val}]$.

In this chapter, the Entropy (Equation (6.1)) and MutualInfo (Equation (6.3)) functions refer to the appropriate entropy-based computation just described, where $p(h)$, $p(o|l)$, and $p(h|o,l)$ are computed using the ModelCount algorithm described in the next section. Using MutualInfo, an attacker can optimize the information gain by trying many different l_{val} values and computing the corresponding MutualInfo. Observe that this process involves model counting for instantiating constraints for many values of l_{val}. In the next section I describe how to perform this model counting step efficiently.

Incremental Constraint Solving and Model Counting

As mentioned above, I compute entropy, which is used in the objective function for information gain, using model counting. For this purpose, I use and extend the Automata-Based Model Counter (ABC) tool, which is a constraint solver for string and numeric constraints with model counting capabilities [46]. Attack synthesis requires solving and model counting the constraint on the secret, C_h, and updating it with the current instantiated observation constraint, $\psi_o[l \mapsto l_{val}]$ (Algorithm 8, line 5). This results in a new constraint, $C_h \wedge \psi_o[l \mapsto l_{val}]$, which I then compute the entropy for (Algorithm 8, line 6). As this process is executed many times, multiple calls to ABC are required, often with similar constraints. In each iteration ABC starts from scratch re-solving each sub-constraint again and constructing a DFA for each of them, then combining them using DFA intersection. Note that, during attack synthesis, C_h can become a complex combination of constraints that represent what I learned over the course of the attack. Then ABC would be unnecessarily re-solving the subconstraints of C_h in each attack step. To summarize, I observe that, during attack synthesis 1) the constraint that characterizes the set of secrets that are consistent with the observations and low inputs (C_h) is constructed incrementally, and 2) computing entropy using incremental constraint solving can improve the performance by exploiting the incremental nature of attack synthesis.

I implemented incremental constraint solving and model counting by extending ABC so that it retains state over successive calls. Given a constraint, ABC constructs the automaton representing the set of solutions to the constraint, which is then stored for use in later calls. The steps of attack synthesis involve two types of model counting for C_h and $\psi_o[l \mapsto l_{val}]$: during MutualInfo when an attacker optimizes the attack by trying many different l_{val}, and in Entropy, during computation of the remaining uncertainty. In both situations, model counting is required on many different constraints, and most of the sub-constraints come from previous iterations. I augmented ABC with an interface so that, given $C_h \wedge \psi_o[l \mapsto l_{val}]$, I can check if an automaton has already been constructed for either C_h or $\psi_o[l \mapsto l_{val}]$, and if so, to get the already constructed automata for them, rather than re-solving each constraint. Note that for the purposes of model counting, $\psi_o[l \mapsto l_{val}]$ can be represented as $\psi_o \wedge l = l_{val}$. My incremental model counting approach is outlined in Algorithm 9. Given the constraint $C_h \wedge \psi_o \wedge l = l_{val}$, GetDFA retrieves the previously constructed automaton for C_h, A_{C_h}. Algorithm 9 is called with a new observation constraint ψ_o in each attack step, for which the automaton must first be constructed. Subsequent calls with the same ψ_o use the previously constructed automaton. A new $A_{l=l_{val}}$ must be constructed for each model counting query (as each query involves a different l_{val}). The final automaton A is constructed using automata product from $A_{C_h}, A_{\psi_o}, A_{l=l_{val}}$. A is exactly the same automaton constructed from $C_h \wedge \psi_o \wedge l = l_{val}$, but it is constructed incrementally, thus allowing re-use of previously constructed automata.

Algorithm 9 ModelCountIncremental($C_h \wedge \psi_o \wedge l = l_{val}$)
Performs incremental model counting for constraint $C_h \wedge \psi_o \wedge l = l_{val}$.

```
1: $A_{C_h} \leftarrow$ GetDFA($C_h$)
2: if IsConstructed($\psi_o$) then
3:     $A_{\psi_o} \leftarrow$ GetDFA($\psi_o$)
4: else
5:     $A_{\psi_o} \leftarrow$ Construct($\psi_o$)
6: $A_{l=l_{val}} \leftarrow$ Construct($l = l_{val}$)
7: $A \leftarrow A_{C_h} \cap A_{\psi_o} \cap A_{l=l_{val}}$
8: return ModelCount($A$)
```

6.4 Attack Synthesis Heuristics

At every attack step the attacker's goal is to choose a low input l^* that reveals information about h^*. Here I will describe techniques based on constraint solving and meta heuristics for synthesizing attack inputs l^*. Meta heuristic approaches explore a subset of the possible low inputs. In order to search the space efficiently, I first observe that I need to restrict the search to those l that are consistent with C_h, which I now discuss.

Constraint-based Model Generation of Low Inputs. The first l value can be chosen arbitrarily since initially I do not have any information about the secret h. After the first step, My attack synthesis algorithm maintains a constraint C_h which captures all h values that are consistent with the observations so far (Algorithm 8, line 5). Using the observation constraints Ψ (which identify the relation among the secret h, public input l and the observation o), I project C_h to a constraint on the input l, which I call C_l, and I restrict My search on l to the set of values allowed by C_l. I.e., I only look for l values that are consistent with what I know about h (which is characterized by C_h) with respect to Ψ. This approach is implemented in GetNeighborInput function which returns an l_{val} by mutating the previous l_{val}.

Searching via Random Model Generation. As a base-line search heuristic, I make use of the approach described above for generating low values that are consistent with C_h. The simplest approach is to generate a single random model from C_l and use it as the next attack input. I call

Algorithm 10 AttackInput-SA(C_h, Ψ)
Generates a low input at each attack step via simulated annealing.

```
1: $t \leftarrow t_0$, $l_{val} \leftarrow$ GetInput($\Psi, C_h$), $\mathcal{I} \leftarrow$ MutualInfo($\Psi, C_h, l_{val}$)
2: **while** $t \geq t_{min}$ **do**
3:     $l_{val} \leftarrow$ GetNeighborInput($l_{val}, \Psi, C_h$)
4:     $\mathcal{I}_{new} \leftarrow$ MutualInfo($\Psi, C_h, l_{val}$)
5:     **if** $(\mathcal{I}_{new} > \mathcal{I}) \vee \left(e^{(\mathcal{I}_{new} - \mathcal{I})/t} > \text{RandomReal}(0,1)\right)$ **then**
6:         $\mathcal{I} \leftarrow \mathcal{I}_{new}, l^* \leftarrow l_{val}$
7:     $t \leftarrow t - (t \times k)$
8: **return** $l^*$
```

this approach Model-based (M). A slightly more sophisticated approach is to generate random samples using C_l, compute the expected information gain for each of them using Equation (6.3) (i.e., objective function is evaluated using the automata-based entropy computation) and then choose the best one. [43] evaluates different meta heuristic techniques : genetic algorithm (GA) and simulated annealing (SA) to maximize information leakage and shows that SA performs better than GA. The reason is GA applies mutation and crossover to generate candidate low values. To restrict the search to l values that are consistent with C_l, would require implementing mutation and crossover operations with respect to C_l. I am not aware of a general approach for doing this, so during GA-based search, mutation and crossover operations can generate low values that are inconsistent with C_l (and hence C_h). Note that, such values will have no information gain and will be ignored during search, but they can slow down the search increasing the search space and hence, I may end up having a higher number of attack steps compared to SA. So, the SA ends up being a more effective meta-heuristic for attack synthesis.

Simulated Annealing. Simulated annealing (SA) is a meta-heuristic for optimizing an objective function $g(s)$ [112]. SA is initialized with a candidate solution s_0. At step i, SA chooses a neighbor, s_i, of candidate s_{i-1}. If s_i is an improvement, i.e., $g(s_i) > g(s_{i-1})$, then s_i is used as the candidate for the next iteration. If s_i is not an improvement, i.e. $g(s_i) \leq g(s_{i-1})$, then s_i is still used as the candidate for the next iteration, but with a small probability p. Intuitively, SA is a controlled random search that allows a search path to escape from local optima by permitting the search to sometimes accept worse solutions. The acceptance probability p decreases exponentially over time, which is modeled using a search "temperature" which "cools off" and converges to a steady state. My use of SA that incorporates automata-based entropy computation is given in Algorithm 10 where I use GetNeighborInput function to get new candidates.

6.5 Implementations and Experiments

Implementation. The implementation of my approach consists of two primary components, corresponding to the two main phases described in section 6.2. I implement Algorithm 7 using Symbolic Path Finder (SPF) [110]. I implement Algorithm 8 as a Java program that takes the observation constraints generated by Algorithm 7 as input, along with C_h, h^*. AttackInput-SA from section 6.4 is implemented directly in Java as well. I implement GetNeighborInput, ModelCount, and ModelCountIncremental by extending the existing string model counting tool ABC as described in section 6.3. I add these features directly into the C++ source code of ABC along with corresponding Java APIs.

Table 6.7: Benchmark details with the number of path constraints ($|\Phi|$) and the number of merged observation constraints ($|\Psi|$).

Benchmark	ID	Operations	Low Length	High Length	$\lvert\Phi\rvert$	$\lvert\Psi\rvert$
passCheckInsec	PCI	charAt,length	4	4	5	5
passCheckSec	PCS	charAt,length	4	4	16	1
stringEquals	SE	charAt,length	4	4	9	9
stringInequality	SI	$<,\geq$	4	4	2	2
stringConcatInequality	SCOI	concat,$<,\geq$	4	4	2	2
stringCharInequality	SCI	charAt,length,$<,\geq$	4	4	80	2
indexOf	IO	charAt,length	1	8	9	9
compress	CO	begins,substring,length	4	4	5	5
editDistance	ED	charAt,length	4	4	2170	22

Benchmark Details. To evaluate the effectiveness of My attack synthesis techniques, I experimented on a benchmark of 9 Java canonical programs utilizing various logical and string manipulation operations, setting different sizes and lengths to define the domain of secret value

(Table 6.7). The functions PCI and PCS are password checking implementations. Both compare a user input and secret password but early termination optimization (as described in chapter 6) induces a timing side channel for the first one and the latter is a constant-time implementation. I analyzed the SE method from the Java String library which is known to contain a timing side channel [113]. I discovered a similar timing side channel in the IO method from the Java String library. Function ED is an implementation of the standard dynamic programming algorithm to calculate minimum edit distance of two strings. Function CO is a basic compression algorithm which collapses repeated substrings within two strings. SI, SCOI and SCI functions check lexicographic inequality $(<, \geq)$ of two strings whereas first one directly compares the strings, second one includes *concat* operation with inequality and third one compares characters in the strings.

Experimental Setup. For all experiments, I use a desktop machine with an Intel Core i5-2400S 2.50 GHz CPU and 32 GB of DDR3 RAM running Ubuntu 16.04, with a Linux 4.4.0-81 64-bit kernel. I used the OpenJDK 64-bit Java VM, build 1.8.0 171. I ran each experiment for 5 randomly chosen secrets. I present the mean values of the results in Table 6.8. For SA, I set the temperature range (t to t_{min}) from 10 to 0.001 and cooling rate k as 0.1.

Results. In this discussion, I describe the quality of a synthesized attack according to these metrics: attack synthesis time, attack length, and overall change in uncertainty about the secret measured as entropy from $\mathcal{H}_{init}$ to $\mathcal{H}_{final}$ and efficiency of incremental attack synthesis in terms of time. Attacks that do not reduce the final entropy to zero are called *incomplete*. Incomplete attacks are mainly due to one of two reasons: the program is not vulnerable to side-channels (for example PCS) or the observation constraints are very complex, combining lots of path constraints which slows progress too much so that not enough information is leaked within the given time bound (for example ED and SCI). For the purpose of direct comparison, in My experiments, I set a bound of 5 hours for SA (slowest technique) on ED and SCI and computed a bound for $\mathcal{H}_{final}$ of **17.28** and **14.48**, respectively, while all other examples reduced $\mathcal{H}_{final}$ to **0.0**.These examples are marked with $*$. Note that, M and SA-I techniques can reduce $\mathcal{H}_{final}$ for ED and

SCI to **14.34** and **12.28**, respectively, after one hour.

Attack Synthesis Time Comparison. I observe that the model-based technique (M), which only uses C_l to restrict the search space is faster than other techniques, as it greedily uses a random model generated by ABC as the next attack input, with no time required to evaluate the objective function. M quickly generates attacks for most of the functions. I examined those functions and determined that their objective functions are "flat" with respect to l. Any l_{val} that is a model for C_l at the current step yields the same expected information gain. Figure 6.4 shows how M can synthesize attacks faster compared to SA (in seconds).

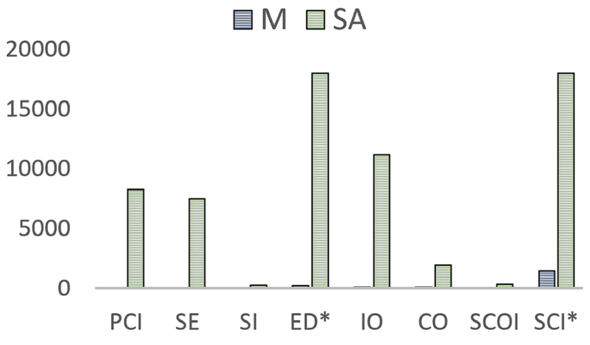


Figure 6.4: Synthesis Time, M vs SA

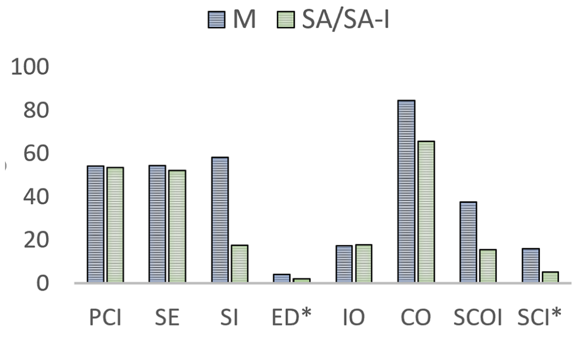


Figure 6.5: Attack Length, M vs SA/SA-I

Attack Length Comparison. Although M is fast in synthesizing attacks and generates attacks for each benchmark, experimental results show that it requires more attack steps (in terms of information gain) compared to the attacks generated by meta-heuristic techniques that optimize the objective function. As the experimental results show for the SI, SCOI and SCI, a meta-heuristic technique can reduce $\mathcal{H}_{final}$ further but with fewer attack steps compared to the model-based approach (M). And, this case would be true for any example where different inputs at a specific attack step have different information gain. If attacker is aware of the "flat" objective function phenomenon, they can proceed with M. In general, M is not efficient to generate an attack with reduced number of attack steps and hence, meta heuristics like SA approach are required. Figure 6.5 shows how SA is better than M in terms of length of the generated attacks. Note that, I say M vs SA/SA-I as incremental version will make difference in attack synthesis time, not attack length.

Efficiency of Incremental Attack Synthesis. On one hand, I can synthesize attacks faster using M but attacks synthesized by M require more attack steps in general. On the other hand, I can synthesize attacks with minimal number of attack steps using SA, but attack synthesis process is slower for SA. My experiments demonstrate that incremental attack synthesis using SA gives us fast attack synthesis without increasing the attack length. I compare incremental version of SA (SA-I) against SA. Figure 6.6 shows SA-I is an order of magnitude faster than SA for all the examples from the benchmark. I also compare SA-I against M and Figure 6.7 shows that SA-I is comparable to M in terms of attack synthesis time (in seconds).

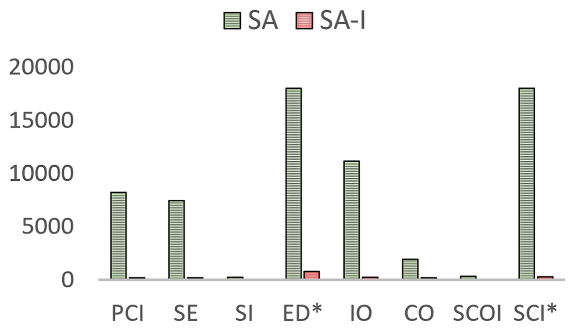


Figure 6.6: Synthesis Time, SA vs SA-I

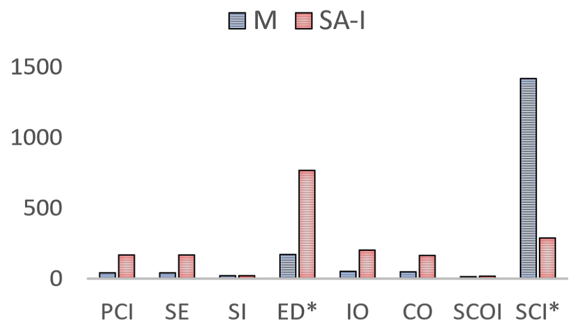


Figure 6.7: Synthesis Time, M vs SA-I

Table 6.8: Experimental results for secure password checker (PCS). Time bound is set as 3600 seconds.

ID	$\mathcal{H}_{init}$	Metrics	M	SA	SA-I
PCS	18.8	Steps	108	14	99
		$\mathcal{H}_{final}$	18.8	18.8	18.8

Vulnerability to Side-Channels. Finally, I observe that some of My selected benchmarks are more secure against My attack synthesizer than others. In particular, `PCS`, a constant-time implementation of password checking, did not leak any information through the side channel. One of the examples from the benchmark, `ED` also did not succumb to My approach easily, due to the relatively large number of generated constraints (2170), indicating a much more complex relationship between the inputs and observations. To summarize, My experiments indicate that My attack synthesis approach is able to construct side-channel attacks, providing evidence of vulnerability (e.g. `PCI`). Further, when attack synthesizer fails to generate attacks (`PCS`), or is only able to extract a relatively small information after many steps of significant computation time (`ED`), it provides evidence that the function under test is comparatively safer against side-channel attacks. Table 6.8 shows results for `PCS`, hardly reducing $\mathcal{H}_{\mathit{final}}$ even after running for 1 hour for M, SA and SA-I.

6.6 Case Studies

My experimental results show that synthesizing attacks face scalability issues for programs leading to large numbers of complex observation constraints. Note that, this limitation depends on the limitations of the building blocks: constraint solvers and model counters. The more powerful these tools become, more powerful attack synthesis will be. I now present two case studies.

CRIME Attack. The "Compression Ratio Info-leak Made Easy" (CRIME) attack [105] allows an attacker to learn fields of encrypted web session headers by injecting extraneous text (l) into a procedure that compresses and encrypts the header (h). Despite the encryption, an attacker can infer how much of the injected text matched the unknown header by observing the number of bytes in the compressed result [39, 105]. My approach automatically synthesizes this attack. Symbolic execution of the compression function (LZ77T) for a secret of length 3 and alphabet size 4 yields 187 path constraints and 4 observations, leading to 4 observation constraints. M synthesizes an attack in 6.8 steps within 468.5 seconds. SA-I could generate the attack in 7.8 steps within 757.4 seconds. SA-I does not improve over M due to "flat" objective function. Note that [39] performs leakage quantification for this example but does not synthesize attacks.

Law Enforcement Database. The Law Enforcement Employment Database Server is a network service application included as a part of the DARPA STAC program [25, 114]. This application provides access to records about law enforcement personnel. Employee information is determined by an employee ID number. The database contains restricted and unrestricted employee information. Users can search ranges of employee IDs. If an ID query range contains one or more restricted IDs, the returned data will not contain the restricted IDs. I decompiled the application and then symbolically executed the `channelRead0` method from the `UDPServerHandler` class which performs the database search operation. I limited the domain of ids to 1024, added 30 unrestricted IDs and 1 restricted ID. Symbolic execution gives us 1669 path constraints with 162 distinguishable observations ($\delta = 10$ instructions). M generates attack with an attack length of 8.2 in 270.1 seconds whereas SA-I generates an attack with length of 6.5 in 810.7 seconds. SA-I requires less attack length as the objective function is not “flat”.

Chapter 7

Related Works

There has been a large number of works on the development of automated testing techniques and quantitative information flow analysis. In this chapter, I discuss all the existing works, related to the contributions I provide in this thesis.

Symbolic Quantitative Software Analysis.

There has been an increasing amount of research on quantitative program analysis techniques based on model counting constraint solvers, and there has been a surge of progress in model counting constraint solvers [46, 55–58, 115]. Model counting constraint solvers have been used in a variety of quantitative program analysis tasks such as probabilistic analysis [19, 21, 116], reliability analysis [62], estimating performance distribution [117], s quantitative information flow [30, 34, 39, 118, 119], and side-channel attack synthesis [40, 44, 65]. Branch selectivity and probabilistic reachability heuristic I introduce in this thesis are fundamental quantitative program analysis techniques and rely on the recent developments in model counting constraint solvers.

Probabilistic symbolic execution [19] and statistical symbolic execution [21] can be used for probabilistic reachability analysis problem I approach in this thesis. However probabilistic symbolic execution suffers from path explosion [10] and increasing size of path constraints with increasing execution depth, which can lead to double exponential blow up. Moreover, probabilistic symbolic execution can only analyze program behaviors up to a fixed execution depth. Statistical symbolic execution [21] is more efficient compared to probabilistic symbolic execution but still suffers with increasing execution depth. The approach for probabilistic reachability analysis I present in this thesis using branch selectivity addresses these issues since it does not suffer from path explosion and it analyzes branch conditions instead of path constraints modeling behaviors of arbitrarily long paths.

Fuzz Testing

Mutation-based coverage guided Fuzzers. AFl [1] is the vanilla mutation-based coverage guided fuzzer. AFL++ [87] is the latest version of AFL with more speed, better mutation techniques, better instrumentation and support from custom modules. In this work, I use default version of AFL++ which uses power schedule of AFLFast [79]. There are a lot of mutation-based coverage guided fuzzers focusing on advanced mutation strategies. MOPT [82] focuses on mutation scheduling by providing different probabilities to the mutation operators. LAF-INTEL [120] focuses on bypassing hard multibyte comparisons, by splitting them into multiple single-byte comparison. REDQUEEN [80] focuses on bypassing Input-To-State (I2S) defined comparisons. I2S is a type of comparison having a direct dependency with the input in at least one of its operand. Steelix [121] performs static analysis and extra instrumentation to produce inputs satisfying multi-byte comparisons. VUzzer [81] identifies input positions used in the comparison and immediate values using a Markov Chain model and decides which parts of the program should be targeted. FairFuzz [4] identifies the rare branches in the program based on the hitcounts of branches. If a rare branch is identified by FairFuzz, it applies input mutation masking to keep the input part for the rare branches fixed and modify later parts of inputs to explore more deeper branches. In this thesis, I focus on rare program paths, not rare branches. I neither use a fuzzer to identify rare branches nor modify mutation strategies inside the fuzzer. I do not make any changes to the fuzzing technique. I generate a set of rare inputs for fuzzer using a lightweight static analysis technique.

Hybrid Testing. Hybrid testing techniques [6, 11–14] combine concrete and symbolic techniques in order to improve effectiveness of testing. Strategy function for hybrid testing need to decide when to apply concrete techniques and when to apply symbolic. Existing techniques assess the difficulty of concrete testing to do make the decision based on the saturation of random testing [11, 13] or using a predefined configuration of time to run for concrete and symbolic techniques [12] or probabilistic program analysis [6, 14]. Markov decision process construction extracting control flow graph and putting probabilities as edge weight has been used to find optimal strategy for concolic testing [14]. Probabilistic path prioritization is used in [6] to decide when to invoke symbolic execution in hybrid fuzzing. My approach focuses on identifying *hard to reach* statements based on probabilistic reachability heuristic.

Symbolic execution guided fuzzers. Fuzzing techniques [6, 11–14] use symbolic execution and constraint solvers to generate inputs to pass complex checks in the program. Driller [83] uses selected symbolic execution when fuzzer can not cover new branches for a long period of time. DigFuzz [6] uses the fuzzer itself to statistically identify hardest paths for the fuzzer to explore and then uses symbolic execution to solve path constraints for the hardest paths. Every time it identifies that a branch is hard to pass, invokes symbolic execution. DeepFuzzer [122] uses lightweight symbolic execution to pass initial complex checks and then depend on seed selection and mutation techniques. In this work, I do not use the path samples from fuzzer to identify rare paths rather statically analyze programs. Moreover, I do not symbolically execute the whole program rather guide it using the rare paths I identify.

Grammar-based Fuzzers Grammar-based fuzzing techniques generate well-formed inputs based on a user provided grammar [123, 124]. These fuzzing techniques mutate inputs using the derivative rules in the grammar. As a result, the mutated input is also guaranteed to be well-formed [125]. Grammar-based fuzzers are very effective to fuzz programs that are heavily dependent on structured inputs [123,126]. However, grammar-based fuzzers require application specific knowledge of the program under test. There are several fuzzers [90,92,127–129] focusing specifically on structured inputs such as fuzzing network protocols, compilers, parser for json, xml, xslt files etc. Compared to grammar based fuzzing, the technique I provide is general, it does not require any knowledge about the program under test and it is fully automated. I neither need to provide an input grammar, nor feed inputs to the parser [90,126] or collect large data samples [92] like techniques that specialize on structured inputs.

Seed generation for fuzzers. There are fuzzing techniques that focuses on seed selection and seed prioritization to improve fuzzing efficiency [130–132]. SpotFuzz [130] identifies invalid execution and time consuming edges as hot spots based on hitcounts of different inputs on the edges and proposes a fuzzing solution to reduce energy waste. SLF [131] is a technique which focuses on valid seed input generation. It starts with a very short random input for fuzzing and then performs sophisticated input mutation to get through the validity checks. [132] systematically investigates and evaluates the affect of seed selection on fuzzer's ability to find bugs and demonstrates that fuzzing outcomes vary depending on the initial seeds used. In this work, I also demonstrate that rare inputs as initial seeds bootstrap the fuzzer. However, instead of focusing on corpus minimization techniques or generating valid inputs [131] or seed selection algorithms [130] I focus on generating seeds that can execute rare paths. *Static program analysis for fuzzing.* A huge number of fuzzing techniques [80, 81, 90, 121, 133, 134] use static program analysis techniques to guide fuzzers. Most of these techniques use either control flow analysis or taint analysis. In this work, I also use control flow analysis and dependency analysis to identify rare paths. I introduce a new kind of control flow paths (II-paths) for rare path analysis. My concept behind II-paths are inspired from the control flow directed concolic search techniques provided in [15].

Quantitative Information Flow Analysis

Quantitative measurement of information leakage has been an active area of research. Early work [27] measured the number of tainted bits which provides a coarse approximation of information leakage. Channel capacity has been used to quantify information leakage in programs [28–32]. Channel capacity makes worst case assumptions about the probability distributions of the inputs. It has been used in [28] to measure influence of the inputs to correctly distinguish false positives by taint analysis from the real attacks. A recent work [31] uses channel capacity to quantify information leakage and provides an automated and scalable approach for computing it. In this thesis, I show that Shannon entropy-based measure of information leakage provides a more precise and informative measure. Channel capacity have also been used for computing information leakage in concurrent probabilistic programs [32]. In this thesis I focus on deterministic programs only.

A large number of works [30, 33–37] use information theoretic approaches for quantifying information leakage. There are previous works that use Shannon entropy-based information leakage computation [38–44]. The technique in [38] is based on an enumeration algorithm. Other works [39–44] use model counting techniques [45–47] to compute information leakage. Constraints that are supported in these techniques are linear arithmetic constraints. The approach in [40] also relies on generation of a closed form objective function that represents the information leakage, and uses model counting techniques that specialize on linear integer arithmetic to construct such a function. None of these techniques support non-linear constraints. I support both linear and non-linear constraints using ABC [46] and SearchMC [49]. Moreover, all these model counting tools provide exact count and hence their technique for quantifying information leakage is straight-forward. In this thesis, I use an approximate model counting tool [49] to support model counting for bit-vector constraints and hence can cover quantitative analysis for programs with wide range of computations and generated constraints.

LeakWatch [135] estimates leakage in Java programs by sampling program executions on concrete inputs. An earlier work [42] applies sampling of symbolic paths and provide anytime bound under- and over-approximating the exact leakage to make their technique scalable. This work measures precise leakage for probabilistic programs considering noise and randomness in program execution and observations and provides techniques to compute anytime leakage considering incomplete path coverage by symbolic execution. But, they do not use any approximate model counter. In this thesis, I provide techniques to compute exact or sound bound on leakage with exact and approximate model counting considering full or incomplete path coverage by a symbolic execution tool.

Side-Channel Attack Synthesis

There are two previous results that are most closely related to my work on attack synthesis [39, 104]. The first focuses on quantifying information leakage through side channels for programs manipulating strings [39]. This work assumes that the given program has a segment oracle side-channel vulnerability and then quantifies the amount of information leakage for that vulnerability. Other recent work synthesizes side-channel attacks using either entropy-based or SAT-based objective functions, but works only for linear arithmetic and bit-vector constraints [104], using model counters and constraint solvers for those theories [136]. This earlier approach also relies on generation of a closed form objective function that represents the information leakage, and uses model counting techniques that specialize on linear integer arithmetic to construct such a function. In contrast, My approach is more general and can handle any program with numeric, string and mixed constraints. Furthermore, My approach does not require a closed form solution for the objective function as I use meta heuristics to search for input values that leak maximum information. Both of these earlier approaches use constraint solving and model counting queries to quantify the information leakage, but they do not use an incremental approach and, therefore, re-compute many sub-queries.

There are many works on analyzing side-channels in various settings [39, 95, 137–140]. A few recent works address either synthesizing attacks or quantifying information leakage under a model where the attacker can make multiple invocations of the system [38,39,65,95,139]. Single-run analysis is addressed in [141] where bounded model checking is used over the k-composition of a program to determine if it can yield k different outputs. Further, LeakWatch [135] estimates leakage in Java programs based on sampling program executions on concrete inputs. There has been work on multi-run analysis using enumerative techniques [38]. None of these earlier results present a symbolic and incremental approach to adaptive attack synthesis as I present in this thesis.

Due to the importance of model counting in quantitative program analyses, model counting constraint solvers are gaining increasing attention [46,55,142]. ABC is the only one that supports string, numeric and mixed constraints. I extended ABC to perform incremental model counting for my attack synthesis approach. Other work in quantifying information leakage [65,95,104,143] have used symbolic execution and model-counting techniques for linear integer arithmetic.

Chapter 8

Conclusions

Building dependable and secure computing systems is one of the major concerns for software development. Even after the surge of developments focusing on software quality assurance and security, there are remaining challenges. Most well-known and effective approach of assuring software quality is automated testing. It is crucial to develop techniques that can cope up with the existing challenges of automated testing. In this thesis, I focus on the challenges of automated testing by analyzing software programs in a quantitative manner. To be specific, I develop symbolic quantitative program analysis techniques, focusing on the assessment of testing difficulty and guidance for the testing techniques.

Moreover, software systems that leak private and sensitive information have become a threat in recent years. Software systems vulnerable to side-channels leak information, hence, unauthorized users can gain access to secret information. It is crucial to quantify the amount of information that is leaked in order to measure the threat. In this thesis, I focus on developing symbolic quantitative program analysis techniques to quantify how much information a software system can leak. I also develop techniques to synthesize side-channel attacks, demonstrating vulnerability of a software to side-channel.

Towards assessing testing difficulty, I present a novel heuristic for probabilistic reachability analysis to identify hard to reach program statements given that the program is being tested on random inputs. I use dependency analysis, model counting, abstract interpretation, and probabilistic model checking to compute probability of reaching a program statement. I experimentally evaluate the technique on a set of benchmark programs and results show that the approach can identify hard to reach statements with high precision and accuracy. I provide detailed comparison of the technique against probabilistic symbolic execution and statistical symbolic execution (existing other techniques that can be used to identify hard to reach statements), demonstrating efficiency of the the heuristic approach I develop.

To guide software testing, I provide heuristic to identify rare program paths that are difficult for a fuzzing technique to explore generating random inputs. To identify the rare paths, I use lightweight static analysis based on control flow analysis, dependency analysis and branch model counting. Then, I use the identified rare paths to guide a concolic execution tool to generate inputs that can execute these rare paths. Finally, I provide these inputs as the initial seed set to the fuzzer. From the experimental evaluation on a set of benchmarks, heavily dependent on structured inputs, I find that it is possible to generate inputs that a fuzzer cannot generate mutating inputs given a time budget. These inputs also guide the fuzzer to achieve better coverage compared to initial random seed. To speed up the rare path analysis, I also introduced a new type of control flow paths (II-paths) in this chapter. Experimental evaluation demonstrates that II-paths are more efficient to generate inputs exploring rare paths compared to intra-procedural paths and inter-procedural paths given a time budget.

Towards quantifying information leakage, I build techniques on top of existing quantitative program analysis techniques for computing sound information leakage bounds for software. My approach is based on symbolic execution and model counting, and there are two causes of unsoundness in this context: 1) symbolic execution cannot achieve complete path coverage and 2) model counting tools cannot always compute exact count. I present techniques that provide lower and upper bounds for information leakage considering these two sources of unsoundness.

Experimental results show that it is feasible to compute sound bounds for information leakage.

Towards synthesizing side-channel attacks, I generalize the techniques for synthesizing attack using metaheuristic search techniques. The techniques I develop can synthesize attack not only for integers but also for string manipulating programs. I also provide incremental analysis to exploit the inherent iterative nature of the technique. Experimental evaluation shows the effectiveness of the attack synthesis approach on several library functions. Experimental results also demonstrate that the incremental approach improves the efficiency of the technique.

Overall, in this thesis, I focus on symbolic quantitative analysis approaches towards advancing the development of automated testing techniques and quantitative information flow analysis. The techniques I present show promising results, and I believe that the impact of symbolic quantitative analysis techniques in improving dependability and security of software systems will continue to grow in the future.

Bibliography

[1] Michał Zalewski, "American Fuzzy Lop." `http://lcamtuf.coredump.cx/afl/`, 2014.

[2] R. Padhye, C. Lemieux, and K. Sen, *Jqf: Coverage-guided property-based testing in java*, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 398–401, Association for Computing Machinery, 2019.

[3] C. Pacheco and M. D. Ernst, *Randoop: feedback-directed random testing for Java*, in *OOPSLA 2007 Companion, Montreal, Canada*, ACM, Oct., 2007.

[4] C. Lemieux and K. Sen, *Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage*, pp. 475–485, 09, 2018.

[5] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krugel, and G. Vigna, *Driller: Augmenting fuzzing through selective symbolic execution*, in *NDSS*, 2016.

[6] L. Zhao, Y. Duan, H. Yin, and J. Xuan, *Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing.*, in *NDSS*, 2019.

[7] J. C. King, *Symbolic execution and program testing*, *Communications of the ACM* **19** (1976), no. 7 385–394.

[8] C. Cadar, D. Dunbar, and D. R. Engler, *KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs*, in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224, 2008.

[9] S. Anand, C. Păsăreanu, and W. Visser, *Jpf–se: A symbolic execution extension to java pathfinder*, *Tools and Algorithms for the Construction and Analysis of Systems* (2007) 134–138.

[10] C. Cadar and K. Sen, *Symbolic execution for software testing: three decades later*, *Communications of the ACM* **56** (2013), no. 2 82–90.

[11] R. Majumdar and K. Sen, *Hybrid concolic testing*, in *29th International Conference on Software Engineering (ICSE'07)*, pp. 416–426, IEEE, 2007.

[12] M. Dimjašević, F. Howar, K. Luckow, and Z. Rakamarić, *Study of integrating random and symbolic testing for object-oriented software*, in *International Conference on Integrated Formal Methods*, pp. 89–109, Springer, 2018.

[13] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Driller: Augmenting fuzzing through selective symbolic execution*, in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[14] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, *Towards optimal concolic testing*, in *Proceedings of the 40th International Conference on Software Engineering*, pp. 291–302, 2018.

[15] J. Burnim and K. Sen, *Heuristics for scalable dynamic test generation*, in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 443–446, IEEE, 2008.

[16] K. Sen, D. Marinov, and G. Agha, *CUTE: a concolic unit testing engine for C*, in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pp. 263–272, 2005.

[17] J. Geldenhuys, M. B. Dwyer, and W. Visser, *Probabilistic symbolic execution*, in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pp. 166–176, 2012.

[18] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys, *Statistical symbolic execution with informed sampling*, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 437–448, Association for Computing Machinery, 2014.

[19] J. Geldenhuys, M. B. Dwyer, and W. Visser, *Probabilistic symbolic execution*, in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pp. 166–176, 2012.

[20] C. P. Gomes, A. Sabharwal, and B. Selman, *Model counting*, in *Handbook of satisfiability*, pp. 993–1014. IOS press, 2021.

[21] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys, *Statistical symbolic execution with informed sampling*, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 437–448, Association for Computing Machinery, 2014.

[22] S. Saha, M. Downing, T. Brennan, and T. Bultan, *PREACH: A heuristic for probabilistic reachability to identify hard to reach statements*, in *44th IEEE/ACM 44th*

International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 1706–1717, ACM, 2022.

[23] Competition on Software Verification, "SV-COMP." `https://sv-comp.sosy-lab.org/2020/`, 2020.

[24] Apache Commons Lang. `https://commons.apache.org/proper/commons-lang/`, 2020.

[25] "Space/time analysis for cybersecurity (stac)." `https://www.darpa.mil/program/space-time-analysis-for-cybersecurity`.

[26] M. Kwiatkowska, G. Norman, and D. Parker, *Stochastic model checking*, in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pp. 220–270, Springer, 2007.

[27] S. McCamant and M. D. Ernst, *Quantitative information flow as network flow capacity*, in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, (New York, NY, USA), pp. 193–205, ACM, 2008.

[28] J. Newsome, S. McCamant, and D. Song, *Measuring channel capacity to distinguish undue influence*, in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pp. 73–85, 2009.

[29] G. Smith, *On the foundations of quantitative information flow*, in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, pp. 288–302, 2009.

[30] Q. Phan, P. Malacaria, C. S. Pasareanu, and M. d'Amorim, *Quantifying information leaks using reliability analysis*, in *Proceedings of the International Symposium on Model Checking of Software, SPIN 2014, San Jose, CA, USA*, pp. 105–108, 2014.

[31] C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu, *Precisely measuring quantitative information flow: 10k lines of code and beyond*, in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 31–46, IEEE, 2016.

[32] K. Salehi, J. Karimpour, H. Izadkhah, and A. Isazadeh, *Channel capacity of concurrent probabilistic programs*, *Entropy* **21** (2019), no. 9 885.

[33] D. Clark, S. Hunt, and P. Malacaria, *A static analysis for quantifying information flow in a simple imperative language*, *J. Comput. Secur.* **15** (Aug., 2007) 321–371.

[34] M. Backes, B. Köpf, and A. Rybalchenko, *Automatic discovery and quantification of information leaks*, in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pp. 141–153, 2009.

[35] Q. Phan, P. Malacaria, O. Tkachuk, and C. S. Pasareanu, *Symbolic quantitative information flow*, *ACM SIGSOFT Software Engineering Notes* **37** (2012), no. 6 1–5.

[36] V. Klebanov, N. Manthey, and C. Muise, *SAT-Based Analysis and Quantification of Information Flow in Programs*, in *Quantitative Evaluation of Systems*, vol. 8054 of *Lecture Notes in Computer Science*, pp. 177–192. Springer Berlin Heidelberg, 2013.

[37] Q.-S. Phan and P. Malacaria, *Abstract Model Counting: A Novel Approach for Quantification of Information Leaks*, in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, (New York, NY, USA), pp. 283–292, ACM, 2014.

[38] B. Köpf and D. A. Basin, *An information-theoretic model for adaptive side-channel attacks*, in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007* (P. Ning, S. D. C. di Vimercati, and P. F. Syverson, eds.), pp. 286–296, ACM, 2007.

[39] L. Bang, A. Aydin, Q.-S. Phan, C. S. Pasareanu, and T. Bultan, *String analysis for side channels with segmented oracles*, in *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2016.

[40] Q. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, *Synthesis of adaptive side-channel attacks*, in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pp. 328–342, 2017.

[41] L. Bang, N. Rosner, and T. Bultan, *Online synthesis of adaptive side-channel attacks based on noisy observations*, in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 307–322, April, 2018.

[42] P. Malacaria, M. Khouzani, C. S. Pasareanu, Q.-S. Phan, and K. Luckow, *Symbolic side-channel analysis for probabilistic programs*, in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 313–327, IEEE, 2018.

[43] S. Saha, I. B. Kadron, W. Eiers, L. Bang, and T. Bultan, *Attack synthesis for strings using meta-heuristics*, *SIGSOFT Softw. Eng. Notes* **43** (Jan., 2019) 56–56.

[44] S. Saha, W. Eiers, I. B. Kadron, L. Bang, and T. Bultan, *Incremental attack synthesis*, *ACM SIGSOFT Software Engineering Notes* **44** (2019), no. 4 16–16.

[45] V. Baldoni, N. Berline, J. D. Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu, "Latte integrale v1.7.2." http://www.math.ucdavis.edu/ latte/, 2004.

[46] A. Aydin, L. Bang, and T. Bultan, *Automata-based model counting for string constraints*, in *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pp. 255–272, 2015.

[47] A. I. Barvinok, *A Polynomial Time Algorithm for Counting Integral Points in Polyhedra When the Dimension is Fixed*, *Math. Oper. Res.* **19** (1994), no. 4 769–779.

[48] S. Chakraborty, K. S. Meel, and M. Y. Vardi, *A scalable approximate model counter*, in *International Conference on Principles and Practice of Constraint Programming*, pp. 200–216, Springer, 2013.

[49] S. Kim and S. McCamant, *Bit-vector model counting using statistical estimation*, in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 133–151, Springer, 2018.

[50] R. Hund, C. Willems, and T. Holz, *Practical timing side channel attacks against kernel space aslr*, in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 191–205, IEEE, 2013.

[51] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, *Cacheaudit: A tool for the static analysis of cache side channels*, *ACM Transactions on Information and System Security (TISSEC)* **18** (2015), no. 1 4.

[52] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, *Side channel cryptanalysis of product ciphers*, in *European Symposium on Research in Computer Security*, pp. 97–110, Springer, 1998.

[53] Q.-S. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, *Synthesis of adaptive side-channel attacks.*, *IACR Cryptology ePrint Archive* **2017** (2017) 401.

[54] V. Baldoni, N. Berline, J. Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu, *Latte integrale v1. 7.2*, 2004.

[55] L. Luu, S. Shinde, P. Saxena, and B. Demsky, *A model counter for constraints over unbounded strings*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, p. 57, 2014.

[56] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, *Distribution-aware sampling and weighted model counting for SAT*, in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pp. 1722–1730, 2014.

[57] S. Chakraborty, K. S. Meel, R. Mistry, and M. Y. Vardi, *Approximate probabilistic inference via word-level counting*, in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 3218–3224, 2016.

[58] M. Borges, Q. Phan, A. Filieri, and C. S. Pasareanu, *Model-counting approaches for nonlinear numerical constraints*, in *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings* (C. Barrett, M. Davies, and T. Kahsai, eds.), vol. 10227 of *Lecture Notes in Computer Science*, pp. 131–138, 2017.

[59] A. Aydin, W. Eiers, L. Bang, T. Brennan, M. Gavrilov, T. Bultan, and F. Yu, *Parameterized model counting for string and numeric constraints*, in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 400–410, ACM, 2018.

[60] W. Eiers, S. Saha, T. Brennan, and T. Bultan, *Subformula caching for model counting and quantitative program analysis*, in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2019, San Diego, USA, November 10-15, 2019*, 2019.

[61] T. Kapus, M. Nowack, and C. Cadar, *Constraints in dynamic symbolic execution: Bitvectors or integers?*, in *International Conference on Tests and Proofs*, pp. 41–54, Springer, 2019.

[62] A. Filieri, C. S. Pasareanu, and W. Visser, *Reliability analysis in symbolic pathfinder*, in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pp. 622–631, 2013.

[63] Y.-T. S. Li and S. Malik, *Performance analysis of embedded software using implicit path enumeration*, in *ACM SIGPLAN Notices*, vol. 30, pp. 88–98, ACM, 1995.

[64] T. Bultan, *Quantifying information leakage using model counting constraint solvers*, in *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, pp. 30–35, 2019.

[65] L. Bang, N. Rosner, and T. Bultan, *Online synthesis of adaptive side-channel attacks based on noisy observations*, in *Proceedings of the IEEE European Symposium on Security and Privacy*, 2018.

[66] Competition on Software Testing, "Test-COMP." `https://test-comp.sosy-lab.org/2020/`, 2020.

[67] R. Padhye, C. Lemieux, and K. Sen, *Jqf: Coverage-guided property-based testing in java*, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19). https://doi. org/10.1145/3293882.3339002*, 2019.

[68] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, *Semantic fuzzing with zest*, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 329–340, Association for Computing Machinery, 2019.

[69] C. S. Păsăreanu and N. Rungta, *Symbolic pathfinder: symbolic execution of java bytecode*, in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 179–180, ACM, 2010.

[70] D. Balasubramanian, K. Luckow, C. Pasareanu, A. Aydin, L. Bang, T. Bultan, M. Gavrilov, T. Kahsai, R. Kersten, D. Kostyuchenko, Q.-S. Phan, Z. Zhang, and G. Karsai, *ISSTAC: Integrated Symbolic Execution for Space-Time Analysis of Code*, in *submission*, 2017.

[71] S. Wei, P. Mardziel, A. Ruef, J. S. Foster, and M. Hicks, *Evaluating design tradeoffs in numeric static analysis for java*, in *European Symposium on Programming*, pp. 653–682, Springer, Cham, 2018.

[72] B. Jeannet and A. Miné, *Apron: A library of numerical abstract domains for static analysis*, in *International Conference on Computer Aided Verification*, pp. 661–667, Springer, 2009.

[73] G. Singh, M. Püschel, and M. Vechev, *Fast polyhedra abstract domain*, in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 46–59, 2017.

[74] D. Balasubramanian, Z. Zhang, D. McDermet, and G. Karsai, *Janalyzer: A static analysis tool for java bytecode*, *ISIS* **17** (2017) 104.

[75] I. Watson, *Watson libraries for analysis. wala. sourceforge. net/wiki/index. php, Main Page* (2006).

[76] M. Kwiatkowska, G. Norman, and D. Parker, *Probabilistic symbolic model checking with prism: A hybrid approach*, *International Journal on Software Tools for Technology Transfer* **6** (2004), no. 2 128–142.

[77] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, *Spoon: A library for implementing analyses and transformations of java source code*, *Software: Practice and Experience* **46** (2016), no. 9 1155–1179.

[78] Anonymous, "Sv-comp selected benchmarks." `https://anonymous.4open.science/r/185db610-ab95-4f46-8fd5-09e7c1ae662a/`, 2020.

[79] M. Böhme, V.-T. Pham, and A. Roychoudhury, *Coverage-based greybox fuzzing as markov chain*, *IEEE Transactions on Software Engineering* **45** (2017), no. 5 489–506.

[80] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, *Redqueen: Fuzzing with input-to-state correspondence.*, in *NDSS*, vol. 19, pp. 1–15, 2019.

[81] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, *Vuzzer: Application-aware evolutionary fuzzing.*, in *NDSS*, vol. 17, pp. 1–14, 2017.

[82] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, {*MOPT*}*: Optimized mutation scheduling for fuzzers*, in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1949–1966, 2019.

[83] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Driller: Augmenting fuzzing through selective symbolic execution*, in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[84] F. E. Allen, *Control flow analysis*, *ACM Sigplan Notices* **5** (1970), no. 7 1–19.

[85] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, *Cil: Intermediate language and tools for analysis and transformation of c programs*, in *International Conference on Compiler Construction*, pp. 213–228, Springer, 2002.

[86] "Codeql." `https://codeql.github.com`.

[87] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, *{AFL++}: Combining incremental steps of fuzzing research*, in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[88] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, *{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing*, in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 745–761, 2018.

[89] M. Wu, L. Jiang, J. Xiang, Y. Zhang, G. Yang, H. Ma, S. Nie, S. Wu, H. Cui, and L. Zhang, *Evaluating and improving neural program-smoothing-based fuzzing*, .

[90] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, *Parser-directed fuzzing*, in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 548–560, 2019.

[91] "Calculator." `https://github.com/btmills/calculator`.

[92] J. Wang, B. Chen, L. Wei, and Y. Liu, *Skyfire: Data-driven seed generation for fuzzing*, in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 579–594, IEEE, 2017.

[93] "Docker for afl++." `https://hub.docker.com/r/aflplusplus/aflplusplus`.

[94] F. Wang and Y. Shoshitaishvili, *Angr-the next generation of binary analysis*, in *2017 IEEE Cybersecurity Development (SecDev)*, pp. 8–9, IEEE, 2017.

[95] C. S. Păsăreanu, Q.-S. Phan, and P. Malacaria, *Multi-run side-channel analysis using Symbolic Execution and Max-SMT*, in *Proceedings of the 2016 IEEE 29th Computer Security Foundations Symposium*, CSF '16, (Washington, DC, USA), IEEE Computer Society, 2016.

[96] T. M. Cover and J. A. Thomas, *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[97] S. Guiasu and A. Shenitzer, *The principle of maximum entropy*, *The mathematical intelligencer* **7** (1985), no. 1 42–48.

[98] B. F. Albanna, C. Hillar, J. Sohl-Dickstein, and M. R. DeWeese, *Minimum and maximum entropy distributions for binary systems with known means and pairwise correlations*, *Entropy* **19** (2017), no. 8 427.

[99] K. L. Hoffman, *A method for globally minimizing concave functions over convex sets*, *Mathematical Programming* **20** (Dec, 1981) 22–32.

[100] R. Bagnara, P. M. Hill, and E. Zaffanella, *The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems*, *Science of Computer Programming* **72** (2008), no. 1–2 3–21.

[101] C. Cadar, D. Dunbar, D. R. Engler, *et. al.*, *Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.*, in *OSDI*, vol. 8, pp. 209–224, 2008.

[102] "PyParma." `https://github.com/haudren/pyparma`, 2015.

[103] T. Antopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, *Decomposition instead of self-composition for k-safety*, .

[104] Q. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, *Synthesis of adaptive side-channel attacks*, in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA*, 2017.

[105] J. Rizzo and T. Duong, *The crime attack*, Ekoparty Security Conference, 2012.

[106] J. Kelsey, *Compression and information leakage of plaintext*, in *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, pp. 263–276, 2002.

[107] N. Lawson, "Timing attack in google keyczar library." `https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/`, 2009.

[108] T. Nelson, "Widespread timing vulnerabilities in openid implementations." `http://lists.openid.net/pipermail/openid-security/2010-July/001156.html`, 2010.

[109] J. C. King, *Symbolic execution and program testing*, *Commun. ACM* **19** (July, 1976) 385–394.

[110] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, *Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis*, *Automated Software Engineering* (2013) 1–35.

[111] C. Shannon, *A mathematical theory of communication*, *Bell System Technical Journal* **27** (July, October, 1948) 379–423, 623–656.

[112] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, *science* **220** (1983), no. 4598 671–680.

[113] J. S. Daniel Mayer, "Time trial: Racing towards practical remote timing attacks." `https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf`, 2014.

[114] `https://github.com/Apogee-Research/STAC/tree/master/Engagement_Challenges`.

[115] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, *Effective lattice point counting in rational convex polytopes*, *Journal of Symbolic Computation* **38** (2004), no. 4 1273 – 1302.

[116] M. Borges, A. Filieri, M. d'Amorim, and C. S. Pasareanu, *Iterative distribution-aware sampling for probabilistic symbolic execution*, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pp. 866–877, 2015.

[117] B. Chen, Y. Liu, and W. Le, *Generating performance distributions via probabilistic symbolic execution*, in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pp. 49–60, 2016.

[118] Q. Phan, P. Malacaria, O. Tkachuk, and C. S. Pasareanu, *Symbolic quantitative information flow*, *ACM SIGSOFT Software Engineering Notes* **37** (2012), no. 6 1–5.

[119] D. J. Fremont and S. A. Seshia, *Speeding up smt-based quantitative program analysis*, in *In 12th International Workshop on Satisfiability Modulo Theories (SMT)*, July, 2014.

[120] "laf-intel." `https://lafintel.wordpress.com/`. Accessed: 2018-08-21, 2006.

[121] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, *Steelix: program-state based binary fuzzing*, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 627–637, 2017.

[122] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K. R. Choo, *Deepfuzzer: Accelerated deep greybox fuzzing*, *IEEE Transactions on Dependable and Secure Computing* **18** (2019), no. 6 2675–2688.

[123] P. Godefroid, A. Kiezun, and M. Y. Levin, *Grammar-based whitebox fuzzing*, in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pp. 206–215, 2008.

[124] H. Yoo and T. Shon, *Grammar-based adaptive fuzzing: Evaluation on scada modbus protocol*, in *2016 IEEE International conference on smart grid communications (SmartGridComm)*, pp. 557–563, IEEE, 2016.

[125] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, *Fuzzing: State of the art*, *IEEE Transactions on Reliability* **67** (2018), no. 3 1199–1218.

[126] J. Yan, Y. Zhang, and D. Yang, *Structurized grammar-based fuzz testing for programs with highly structured inputs*, *Security and Communication Networks* **6** (2013), no. 11 1319–1330.

[127] S. Bratus, A. Hansen, and A. Shubina, *Lzfuzz: a fast compression-based fuzzer for poorly documented protocols*, .

[128] X. Yang, Y. Chen, E. Eide, and J. Regehr, *Finding and understanding bugs in c compilers*, in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 283–294, 2011.

[129] C. Holler, K. Herzig, and A. Zeller, *Fuzzing with code fragments*, in *21st USENIX Security Symposium (USENIX Security 12)*, pp. 445–458, 2012.

[130] H. Pang, J. Jian, Y. Zhuang, Y. Ye, and Z. Li, *Spotfuzz: Fuzzing based on program hot-spots*, *Electronics* **10** (2021), no. 24 3142.

[131] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, *Slf: Fuzzing without valid seed inputs*, in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 712–723, IEEE, 2019.

[132] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, *Seed selection for successful fuzzing*, in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 230–243, 2021.

[133] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, {*GREYONE*}*: Data flow sensitive fuzzing*, in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2577–2594, 2020.

[134] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, *A taint based approach for smart fuzzing*, in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 818–825, IEEE, 2012.

[135] T. Chothia, Y. Kawamoto, and C. Novakovic, *Leakwatch: Estimating information leakage from java programs*, in *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II* (M. Kutylowski and J. Vaidya, eds.), vol. 8713 of *Lecture Notes in Computer Science*, pp. 219–236, Springer, 2014.

[136] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, *Effective lattice point counting in rational convex polytopes*, *Journal of Symbolic Computation* **38** (2004), no. 4 1273 – 1302. Symbolic Computation in Algebra and Geometry.

[137] D. Brumley and D. Boneh, *Remote Timing Attacks Are Practical*, in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2003.

[138] S. Chen, R. Wang, X. Wang, and K. Zhang, *Side-channel leaks in web applications: A reality today, a challenge tomorrow*, in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 191–206, IEEE Computer Society, 2010.

[139] P. Mardziel, M. S. Alvim, M. W. Hicks, and M. R. Clarkson, *Quantifying information flow for dynamic secrets*, in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pp. 540–555, 2014.

[140] Q. H. Do, R. Bubel, and R. Hähnle, *Exploit Generation for Information Flow Leaks in Object-Oriented Programs*, in *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, (Cham), pp. 401–415, Springer International Publishing, 2015.

[141] J. Heusser and P. Malacaria, *Quantifying information leaks in software*, in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, (New York, NY, USA), pp. 261–269, ACM, 2010.

[142] M.-T. Trinh, D.-H. Chu, and J. Jaffar, *Model counting for recursively-defined strings*, in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, Proceedings, Part II*, pp. 399–418, 2017.

[143] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. d'Amorim, *Quantifying Information Leaks Using Reliability Analysis*, in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, (New York, NY, USA), pp. 105–108, ACM, 2014.