NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SECURING SOFTWARE SYSTEMS

VIA FUZZ TESTING AND VERIFICATION

HONGXU CHEN

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

2019

SECURING SOFTWARE SYSTEMS
VIA FUZZ TESTING AND VERIFICATION

HONGXU CHEN

School of Computer Science and Engineering

A thesis submitted to Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2019

Abstract

Software security has been growing in importance due to the increasing reliance on various systems such as computers, smartphones, and Internet-of-Things devices. Security weaknesseses, or vulnerabilities, range in various aspects, from the systematic levels such as designing defects, to implementation errors such as language-specific program crashes. The multifaceted nature of vulnerabilities poses significant challenges to secure different systems. In fact, multilevel approaches have to be applied to reveal vulnerabilities or solidify the systems. Generally, there are two categories of approaches to securing a system, namely *testing* and *verification*. The former tries to generate test cases to trigger erroneous behaviors in the underlying programs. The latter models the system in an abstracted description to prove the absence of certain security defects. In this thesis, we will describe our efforts in *greybox fuzz testing* and *type checking based verification* in securing different levels of software systems.

Firstly, we apply *greybox fuzz testing* (or *fuzzing*) to detect implementation vulnerabilities. These vulnerabilities are usually subtle that it is rather complicated to formally verify the absence of the corresponding defects. We target on two levels of vulnerabilities. One is to improve the fuzzing directedness of executing towards specific target program locations to reveal crash induced vulnerabilities such as buffer-overflow, use-after-free, etc. To emphasize existing challenges in directed fuzzing, we propose HAWKEYE to feature four desired properties for directedness. With the statically analyzed results on the target program and its target locations, HAWKEYE applies multiple dynamic strategies for seed prioritization, power scheduling and mutation. The experimental results on various real-world programs showed that HAWKEYE significantly outperforms the state-of-the-art greybox fuzzers AFL and AFLGo in reaching

target locations and reproducing specific crashes. The other scenario is to enhance fuzzing to reveal multithreading-relevant bugs. These bugs may root from logic errors which exhibit abnormal behaviors in the target program. They may or may not cause immediate crashes however implicate certain security defects. We present DOUBLADE, a novel thread-aware technique which effectively generates multithreading-relevant seeds. DOUBLADE relies on a set of thread-aware techniques, consisting of a stratified exploration-oriented instrumentation and two complementary instrumentations, as well as dynamic adaptive strategies based on feedbacks. Experiments on 12 real-world programs showed that DOUBLADE significantly outperforms the state-of-the-art fuzzer AFL in generating high-quality seeds, detecting vulnerabilities, and exposing concurrency bugs. To integrate HAWKEYE, DOUBLADE, and other recently proposed fuzzing techniques, we have built our own general-purpose fuzzing framework, Fuzzing Orchestration Toolkit (FOT). Compared to other fuzzing frameworks, FOT is versatile in its functionalities and can be easily configured or extended for various fuzzing purposes. Till now, FOT has detected 200+ zero-day vulnerabilities in more than 40 world-famous projects, among which 51 CVEs have been assigned.

Secondly, we apply *type checking based verification* to detect those vulnerabilities introduced by designing defects where testing usually fails to reveal. Particularly, we focus on the information leakage vulnerability caused by inter-app communications on an Android system. We propose a lightweight security type system that models permission mechanism on Android, where the permissions are assigned statically and facilitated to enforce access control across applications. We proved the soundness of the proposed type system in terms of non-interference, with which we are able to prove the absence of information leakage among various apps in an Android system. We also presented a deterministic type inference algorithm for the underlying type system.

Finally, based on these attempts we have made so far, we briefly discuss the differences between the two categories of applied approaches in terms of the capability, the practicality, and their potential combinations in securing modern software systems.

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarized materials, and has not been submitted for a higher degree to any other University or Institution.

Jul/22/2019

. . . . . .

Date

. . . . . .

Hongxu Chen

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

Jul/22/2019

. . . . . .

Date

. . . . . .

Yang Liu

Authorship Attribution Statement

This thesis contains materials from 3 published and 1 submitted papers in the following peer-reviewed conferences where I am the first author.

Chapter 3 is published as **Hongxu Chen**, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, Yang Liu, "Hawkeye: Towards a Desired Directed Greybox Fuzzer", CCS '18 Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Pages 2095-2108, http://doi.acm.org/10.1145/3243734.3243849, DOI: 10.1145/3243734.3243849 [1].

The contribution of the co-authors are as follows:

- I co-designed and implemented the methodology, wrote the manuscript draft, and conducted the experiments.
- Yinxing Xue co-designed the methodology and revised the major part of the draft.
- Yuekang Li co-designed the methodology and helped evaluate the experimental results.
- Bihuan Chen co-designed the methodology and helped revise the draft.
- Xiaofei Xie helped revise the draft.
- Xiuheng Wu helped evaluate the experimental results.
- Yang Liu co-designed the methodology and supported the research.

Chapter 4 is *submitted* as **Hongxu Chen**, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, Yang Liu, "DOUBLADE: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs".

The contribution of the co-authors are as follows:

- I co-designed and implemented the methodology, wrote the manuscript draft, and conducted the experiments.
- Shengjian Guo, Yinxing Xue, and Yulei Sui co-designed the methodology and revised the major parts of the draft.
- Cen Zhang helped analyze the experimental results.
- Yuekang Li and Haijun Wang helped revise the paper draft.
- Yang Liu supported the research and revised the paper draft.

Chapter 5 is published as **Hongxu Chen**, Yuekang Li, Bihuan Chen, Yinxing Xue, Yang Liu, "FOT: A Versatile, Configurable, Extensible Fuzzing Framework", ESEC/FSE 2018 Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Pages 867-870, http://doi.acm.org/10.1145/3236024.3264593, DOI: 10.1145/3236024.3264593 [2].

The contributions of the co-authors are as follows:

- I and Yuekang Li implemented the core logic of the FOT fuzzing framework, and wrote the manuscript draft as well as conducted all the experiments.
- Bihuan Chen helped revise the draft.
- Yang Liu co-designed the architecture of the framework.
- Yinxing Xue co-designed the HAWKEYE extension of the framework.

Chapter 6 is published as **Hongxu Chen**, Alwen Tiu, Zhiwu Xu, Yang Liu, "A Permission-Dependent Type System for Secure Information Flow Analysis", 2018 IEEE 31st Computer Security Foundations Symposium (CSF), Oxford, 2018, pp. 218-232. https://doi.org/10.1109/CSF.2018.00023, DOI: 10.1109/CSF.2018.00023 [3]. A technical report is available at https://arxiv.org/abs/1709.09623.

The contribution of the co-authors are as follows:

- I co-designed the type system, proved the soundness of the type system with Alwen Tiu, implemented a prototype of the type system, and wrote the draft.
- Alwen Tiu co-designed the type system, proved the soundness of the type system, and revised the paper draft.
- Zhiwu Xu co-designed the type system, and designed the type inference rules for the proposed type system.
- Yang Liu organized and supported the research.

Jul/22/2019

. . . . . .

Date

. . . . . .

Hongxu Chen

Acknowledgements

Firstly, I would express my sincere gratitude to Prof. Yang Liu and Prof Alwen Fernanto Tiu, for their constant encouragement and guidance during my Ph.D years. Prof. Liu has always been supportive and responsible towards my research and personal life both for my PhD years and the year when I was working as a research assistant. He gave me great freedom to explore research topics and provided as many as possible resources to expand my horizon as a researcher. He is also generous to help me rebuild confidence and overcome difficulties when I was confused about my research. Prof. Tiu's rigorous attitude towards research and the critical thinking have left a deep impression on me and inspired me greatly in tackling intractable research problems. I really appreciate their supervision and have learned much from them.

Secondly, I would like to thank my collaborators for their guidance and support. They have been setting good examples for me. I am impressed by the humility and high morality of Prof. Zhiwu Xu, the rigorous writing style of Prof. Bihuan Chen, the problem discovering capability of Prof. Yinxing Xue, the open-minded spirit of Mr. Yuekang Li, and the passionate spirit of Dr. Xiaofei Xie. I also feel lucky to be able to work with other researchers like Prof. Guozhu Meng, Prof. Lei Ma, Dr. Haijun Wang, Mr. Cen Zhang, Mr. Xiuheng Wu.

Thirdly, I would like to express my thanks to Prof. Jun Sun, Prof Shang-Wei Lin, Prof Jianjun Zhao, and Prof Jianqi Shi. They have helped me go through the difficult days and provided me valuable suggestions. Further, I would like to thank all the friends in Cybersecurity Lab, especially Dr. Yuan Zhou, Dr. Ting Su, Dr. Hao Xiao, Dr. Dongxia Wang, Dr. Junjie Wang, Ms. Xiaoning Du, Mr. Zhengzi Xu, Mr. Ruitao Feng, and many other visiting scholars. Pursuing a Ph.D degree has never been easy, but I have been really enjoying these days in Cybersecurity lab.

Finally, I want to express my utmost gratitude to my family. Having been away from home for studies, I am heavily indebted to my parents and relatives for their unconditional understanding and support over the past years. Their meticulous care will continue as one of my biggest sources of inspiration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations and Challenges

Software systems usually suffer from various kinds of vulnerabilities and weaknesses. These vulnerabilities can cause huge catastrophes from financial property loss to massive personnel casualty. For example, *Heartbleed* (CVE-2014-0160) [4] is a critical vulnerability caused by an implementation defect in the widely used OpenSSL cryptographic software library which can compromise the integrity of communications across the entire Web. Further, it is reported that this may also affect the internal networks of enterprise in the coming years due to the fact that most enterprises do not have a good handle on the SSL encrypted services. Other software weaknesses may lead to more direct tragedies which lead to deaths. For example, the aircraft accident of Ethiopian Airlines Flight 302 in late 2018 is very likely due to the improper functionality of the Maneuvering Characteristics Augmentation System (MCAS) from Boeing 737 Max [5]. 153 lives were lost during this accident, including 149 passengers and 8 crew. Ironically, MCAS is designed to keep the balance of the airplane. The same Boeing 737 Max jet led to totally 189 people's death during Lion Air Flight 610 in 2019, which is also reported to result from the functionality of MCAS.

One interesting phenomenon needs to pay attention to is that it is usually a long time from *the point that the vulnerability was introduced* to *the point when the corresponding software was fully upgraded to the patched versions*. According to "Vulnerability

Review 2018 – Global Trends" [6], only 86% vulnerabilities had a patch available on the day of disclosure in 2017, while in 2016 this percentage was 81%. It is worth noting that some vendors choose to issue major product releases only. For the Heartbleed case, it is also interesting that this vulnerability was introduced in 2012 while was not discovered until 2 years later in 2014. Worse still, it also remains unknown when all the mainstream services will be upgraded to get rid of this vulnerability eventually. For the Boing 737 Max case, it is even unclear whether there is a solution for the remedy at all, despite the fact that its first flight was January 2016, which has passed 3 years.

To avoid the damages brought by the software defects, it is favorable to *disclose these vulnerabilities as early as possible and fix them immediately*. However, the vulnerability root causes essentially vary in different levels, including systematic level designing defects, as well as implementation level program crashes, etc. This poses significant challenges to secure these systems. Two widely applied approaches are *testing* and *verification* [7–11]. However, neither of them can comprehensively reveal security defects in the software systems effectively and efficiently. On one hand, verification systematically proves the absence of certain defects in the system. Due to its complex nature, the programs or the systems that are verified are usually abstracted in an intermediate representation and the focus is a relatively higher level security guarantee. However, modeling the system correctly usually takes considerable time and the applied abstraction may miss subtle issues in the implementation. On the other hand, testing is effective in practice to reveal implementation vulnerabilities with test cases; it checks the different functionalities of the target program by considering different application scenarios. However, there is no guarantee that the system is free of vulnerabilities that are not covered by these tests; more importantly, it usually lacks the insight of the higher level design vulnerabilities. For example, although it is possible for fuzz testing to discover multiple crashes in Android core libraries, it usually cannot reveal certain categories of various security issues regarding privacy at the application framework level and the applications built on top of it [12, 13]. In fact, it is almost impossible for testing techniques to detect certain security issues in Android due to its designing flaw [14].

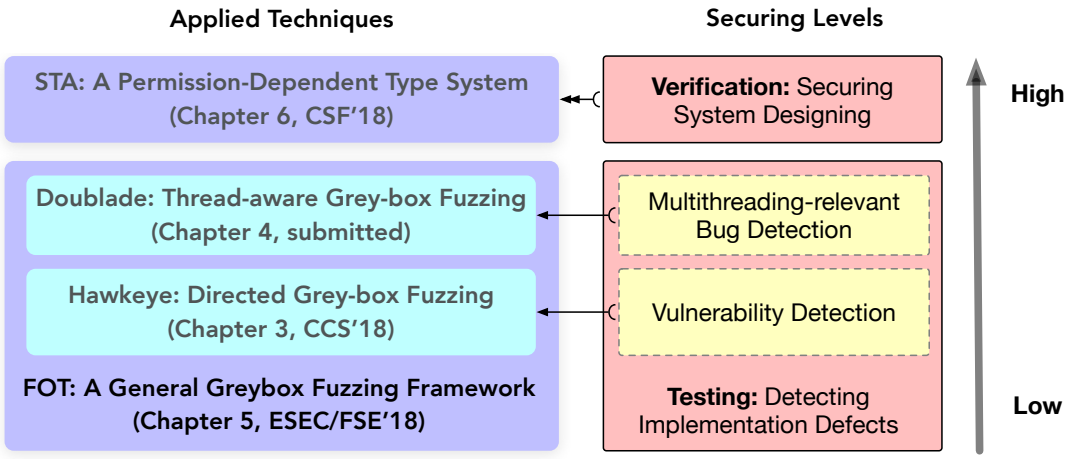

FIGURE 1.1: Main work of this thesis and the relations among its counterparts

1.2 Main Work

My work attempts to secure these systems by applying both testing and verification techniques for different levels of vulnerability detection. Figure 1.1 depicts the major parts of my work as well as their relations between different parts. For lower level implementation vulnerabilities, we rely on testing: this includes detecting vulnerabilities caused by buffer-overflow, use-after-free, etc — this lies in the lowest in the vulnerability hierarchy, as well as detecting relatively higher level concurrency-relevant bugs such as data-races or vulnerabilities in the multithreading environment. For higher level designing defects, we rely on verification, our target is the permission-based access control defects which cannot be (exhaustively) revealed by testing.

As to testing, we apply the greybox fuzzing approaches to reveal those *implementation vulnerabilities*. Firstly, in order to guide the fuzzing procedure to *certain specific program locations*, we propose HAWKEYE, which can be applied to scenarios such as patching testing, crash reproduction, etc. We summarize four key properties that are considered important to the guidance: the robust directedness mechanism, the lightweight yet effective integration with static analysis, the optimized seed prioritization and scheduling strategies, as well as adaptive strategies across fuzzing states. Secondly, in order to improve the efficiency of fuzzing multi-threaded programs, we

propose DOUBLADE, a directed greybox fuzzer which utilizes the thread-aware analyses to increase the quality of the generated seeds. DOUBLADE provides the thread-interleaving feedback by providing a novel stratified exploration oriented instrumentation, together with the threading context information and the thread scheduling intervention that diversifies the actual interleaving. In addition, to integrate these different state-of-the-art fuzzing techniques, we build a general-purpose greybox fuzzing framework called *Fuzzing Orchestration Toolkit* [2] (a.k.a. FOT). This framework aims to be *versatile*, *configurable*, and *extensible*. It is versatile since it has integrated and will integrate various greybox fuzzing techniques [15–25]. It is configurable in the sense that it provides several builtin strategies which allow easy customization for the fuzzing procedure. It is also extensible in the sense that it provides fruitful interfaces for easy integration with new techniques. Both HAWKEYE and DOUBLADE are built on top of FOT, which heavily relies on the latter's static analysis integration.

In order to secure the higher level designing defects such as information leakage in Android system, we propose a permission-dependent security type system to rigorously certify that the underlying system is free of these vulnerabilities. The novelty of the proposed type system is that we designed a type merging operator that allows for permission-dependent typing. We proved the soundness of the type system in terms of *non-interference*, in which we formally assure no information leakage in the system as long as the security type checking is passed. We also provided a set of type inference rules that are guaranteed to be decidable within a finite number of steps. In addition, we implemented a prototype of the type system which can type check the information leakage issues in Android.

With these applied approaches, we aim to provide a comprehensive solution, which ranges in different levels of vulnerability detection, to securing various software systems such as Linux or Android.

1.3 Contributions of the Thesis

The contributions of the thesis are as follows:

1. We proposed a principled directed greybox fuzzing technique, HAWKEYE, to allow for fuzzing on specific program target locations. HAWKEYE provides the robust directedness mechanism, integrates with static analysis to calculate the directedness utilities, optimizes seed prioritization and scheduling, and applies adaptive mutation strategies across different distance states. This significantly improves the fuzzing effectiveness on the specified program target locations.

2. We proposed a thread-aware fuzzing technique, DOUBLADE, which significantly improved the effectiveness of fuzzing on multi-threaded programs. DOUBLADE utilizes the thread-aware analysis and applies three categories of instrumentation to explore thread-interleavings, improve the thread-scheduling diversities across executions, and consider threading context; based on the feedback provided by the instrumentations, DOUBLADE generates more valuable seeds that can significantly help reveal multithreading-relevant bugs.

3. We provided a greybox fuzzing framework, FOT, which is versatile in its configurability and extensibility. Based on this, we have integrated several fuzzing techniques, including HAWKEYE and DOUBLADE, and successfully detected more than 200 vulnerabilities in real-world open source programs, among which 51 have been assigned CVE IDs.

4. We provided a permission-dependent type system that precisely enforces access control based on the permission authentication mechanism on Android and proved its soundness in terms of the non-interference property. We also provided a series of decidable type inference rules to assist the type checking. Based on the type system, we implemented a prototype that stresses information leakage issues in Android system.

1.4 List of Materials Related to the Thesis

1. **Hongxu Chen**, Yuekang Li, Bihuan Chen, Yinxing Xue, Yang Liu, “FOT: A

Versatile, Configurable, Extensible Fuzzing Framework", ESEC/FSE '18, Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Pages 867-870, http://doi.acm.org/10.1145/3236024.3264593, DOI: 10.1145/3236024.3264593.

2. **Hongxu Chen**, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, Yang Liu, "Hawkeye: Towards a Desired Directed Greybox Fuzzer", CCS'18, Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Pages 2095-2108, http://doi.acm.org/10.1145/3243734.3243849, DOI: 10.1145/3243734.3243849.

3. **Hongxu Chen**, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, Yang Liu, "DOUBLADE: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs", submitted.

4. **Hongxu Chen**, Alwen Tiu, Zhiwu Xu, Yang Liu, "A Permission-Dependent Type System for Secure Information Flow Analysis", 2018 IEEE 31st Computer Security Foundations Symposium (CSF'18), Oxford, 2018, pp. 218-232. https://doi.org/10.1109/CSF.2018.00023, DOI: 10.1109/CSF.2018.00023. A technical report is available at https://arxiv.org/abs/1709.09623.

1.5 Outline of the Thesis

The rest of the thesis is organized as follows.

Chapter 2 describes the terminologies and preliminaries throughout this paper. Chapter 3 explains our directed greybox fuzzer HAWKEYE; HAWKEYE is our attempt to guide the fuzzing process to specific program target locations. Chapter 4 describes DOUBLADE, which aims to enhance the fuzzing effectiveness on multithreaded programs. Chapter 5 introduces our greybox fuzzing framework, FOT, which has integrated multiple greybox fuzzing techniques. Chapter 6 describes our attempt to secure

software systems based on verification, which applies a permission-dependent type system to enforce the non-interference property that prevents information leakage in an Android system. Chapter 7 discusses my reflection on the testing and verification based techniques and finally concludes the thesis.

Chapter 2

Preliminaries

In this chapter, we briefly describe the preliminaries, as well as terminologies used throughout the thesis.

2.1 Greybox Fuzz Testing

Since its introduction in the early 1990s [26], *fuzz testing*, or *fuzzing*, has remained highly popular due to its conceptual simplicity, its low barrier to deployment, and its vast amount of empirical evidence in discovering real-world software vulnerabilities [20].

Greybox fuzzers (GBFs), which apply some instrumentations and utilize the collected dynamic statistics as feedback to guide the fuzzing procedure, have been proven to be effective in generating seeds and detecting vulnerabilities in modern programs [20]. In particular, AFL [27], libFuzzer [28], honggfuzz [29] and the fuzzing infrastructure ClusterFuzz [30] have been able to detect more than 16,000 vulnerabilities or bugs in over 160 open source projects [27, 30]. Greybox fuzzing strikes a balance between mutation effectiveness and its execution speed. Compared to blackbox fuzzing, greybox fuzzing, to some extent, is aware of the structure of the target program under test via the instrumentation, therefore it has the capability to effectively guide the fuzzing procedure to much more valuable paths. Compared to white-box fuzzing

Algorithm 1: Greybox Fuzzing

input : Program $\mathbb{P}_o$, Initial input seeds T_S
output: Final seeds T'_S, Vulnerable seeds T_C

```
1  P_f = instrument(P_o) ;                          // instrumentation
2  T_C ← ∅, T'_S ← T_S;
3  while True do
4      t = next_seed(T'_S) ;                        // seed selection
5      M = get_mutation_chance(t) ;                 // power scheduling
6      for i ∈ 1...M do
7          t' = mutated_input(t) ;                  // seed mutation
8          res = run(P_f, t', N_c);                 // calibration execution
9          if is_crash(res) then
10             T_C = T_C ∪ {t'} ;                   // report vulnerable seeds
11         else if cov_new_trace(t', res) then
12             T'_S = T'_S ∪ {t'} ;                 // save "good" seeds
```

techniques such as symbolic execution [31,32], it is extremely lightweight and far more scalable to real-world projects.

Algorithm 1 presents the core procedures of a typical greybox fuzzing algorithm. Basically, it involves an instrumentation step and a fuzzing loop. Given a target program $\mathbb{P}_o$ and input seeds T_S, a GBF first applies the instrumentation to track the coverage information in $\mathbb{P}_o$. That is, the actual target program is $\mathbb{P}_f$. In spite of the difference, since the instrumentation typically does not significantly change the regular flow of a program's semantics, the crash on $\mathbb{P}_f$ usually still indicates a weaknesses inside $\mathbb{P}_o$. In practice, this difference is usually neglected. The instrumentation can be done statically or dynamically. Static source instrumentation is usually applied during the source compilation procedure, with the help of compilation infrastructure such as LLVM [33], GCC [34]. Dynamic binary instrumentation (DBI) is done with the help of DBI frameworks such as Valgrind [35], Intel PIN [36]. Logically, instrumentation does not belong to fuzzing procedure, and its purpose is to collect feedback during fuzzing. Usually, the instrumentation does not involve deep static analyses. The fuzzing loop does the actual mutations and testings against the program.

1. Based on the seed priority, *seed selection* selects the next candidate seed t from the queue which is formed by T_S.

2. *Seed mutation* procedure determines based on previous execution statistics on t to determine how many mutation chances ($\mathbb{M}$) will be provided for t.

3. It enters the *monitored execution* against the variants of t with $\mathbb{M}$ iterations.

    a) Firstly, it applies *seed mutation* on t to generate a seed t'.

    b) Secondly, it goes to the actual *calibration execution* against t'; for statistics collection purpose, the fuzzer usually executes $\mathbb{P}_f$ against t' with continuously $\mathbb{N}_c$ times. This is due to the fact that a single run on a specific seed may not be stable and the fuzzer needs to collect *average statistics* for further scheduling.

    c) Finally, the fuzzer handles the generated seeds based on the execution results: the vulnerable seeds will be reported and those that are considered "good" according to the feedback will be appended to the seed queue for subsequent mutations.

As the fuzzer's awareness of the program structure is based on the instrumentation feedback, a GBF typically does not *know* the target program itself; therefore it usually has no idea when to stop the fuzzing procedure. In practice, however, this is not an issue since the fuzzing procedure is terminated manually or canceled within a time thredshold.

In recent years, many approaches have been proposed to increase the fuzzing effectiveness in different aspects [15, 16, 18–20, 22, 37–40]. For example, Angora [19] proposes to introduce fine-grained instrumentation and applies constraint solving to increase coverages; Coll-AFL [18] proposes a path sensitive approach to avoid collisions; Skyfire [21] attempts to improve the mutation operations on structure-sensitive programs. These have led to further advances in the greybox fuzzing.

2.2 Type Checking Based Verification

2.2.1 Type Checking for Information Flow Analysis

Security type checking in securing information flow is based on the classification of program variables into different security labels. The simplest is to classify some variables as L, meaning low security (public), while other variables as H, meaning high security (private). The goal is to prevent private information variables from being leaked publicly, i.e., prevent information in H variables from *flowing* to L variables.

As to *confidentiality*, the common practice is to treat the set of the security labels as a *lattice* and the information flows only upwards in the lattice [41]. For example, since $L \leq H$, we would allow flows from L to L, from H to H, and from L to H, but we would disallow flows from H to L.

Below is an example taken from Denning [42]. We assume secret: H and leak: L, and the attacker can observe any variable at level L but no variable at H. From the confidentiality requirement, leak:=secret is illegal while the assignment secret:=leak works fine. This is due to the fact in the former case the secret information may be observed by the attacker when he/she sees the value of leak, which has the same value as secret after the assignment and is accessible by the attacker who has an observation level not less than L. The leak is caused by an explicit data flow and is so-called *explicit flow*. On the other hand, control flow can also lead to *implicit flow* leaks. For example,

```
1 if ((secret mod 2) == 0) {
2   leak := 0;
3 } else {
4   leak := 1;
5 }
```

Despite that the if-else construct is not intended for data transfer, this segment is essentially equivalent to leak:=secret, which obviously violates confidentiality. Therefore this is as dangerous as the direct assignment.

Some complex data structures may result in subtle information leaks. For example, if array a is initialized with all 0s, then the program below is still leaking information

due to the fact that after the execution, leak will be assigned to i, which is still the value of secret. There is also a leak from secret to leak.

```
1 a[secret] := 1;
2 for (int i := 0; i < a.length; i++) {
3   if (a[i] == 1) leak := i;
4 }
```

2.2.1.1 The Non-interference Property

At a program level, the security properties are described as *non-interference*, which was described by Goguen and Meseguer in 1982 [43]. The simplest form tells that a deterministic program c satisfies non-interference, if and only if, for any memories μ and ν that agree on L variables, the memories produced by running c on μ and on ν also agree on L variables (provided that both terminate successfully). Basically, the program can be viewed as a state machine. If a low user is working on the machine, it will respond in exactly the same manner (on the low outputs) whether or not a high user is working with sensitive data. The low user will not be able to acquire any information about the activities (if any) of the high user by observing the values of low variables.

Typically, a security type system is introduced to prove the non-interference property. In addition to the regular types (such as integer, boolean, etc.), expressions and commands also contain security labels. And by applying the well-established typing derivation approach, it is fairly enough by reasoning over provided typing rules to verify the property, thus it provides a certificate to the underlying program.

2.2.1.2 Typing Principles

When applying type systems to guard information flows, a security label is typically associated with security type τ apart from the regular data type. The goal is to prove the soundness of the type system; that is, the problem survives the required security properties as long as it is well-typed in terms of the security types.

In Bell-Lapadula Model [44], it requires the subject at a given level must not write to any object at a lower security labels, and not read any object a higher security labels, which is characterized by the phrase "no read up, no write down", and it is safe to treat data that is low confidential to be high. In terms of language based secure type systems, this is termed as *Simple Security* and *Confinement Security*. The former applies to expressions and later applies to commands. If an expression e can be given type τ in the system, then Simple Security says, for secrecy, that only variables at level τ or lower in e will have their contents accessed (read) when e is evaluated ("no read up"). On the other hand, if a command c can be given type τ, then Confinement says, for secrecy, that no variable below level τ is updated (modified, written) in c ("no write down").

The concrete security typing rules vary in the underlying type systems to guarantee these two properties. However, normally two subsumption rules are present in addition to the syntax-directed typing rules. That is, for expressions, it follows a co-variant view(Sub_e); and for commands, it follows a contra-variant view(Sub_c).

$$\frac{e : \tau \qquad \tau \leq \tau'}{e : \tau'}(\text{Sub}_e) \qquad \frac{c : \tau \qquad \tau' \leq \tau}{c : \tau'}(\text{Sub}_c)$$

Intuitively, (Sub_e) depicts the fact that it is always safe to read less confidential data as more confidential one; while (Sub_c) discloses that if there is no variable below level τ updated in c, then there is surely no variable below τ' updated in c, as long as $\tau' \leq \tau$.

2.2.2 IPC Mechanism in Android

Android has become the mainstream of smartphones. Android security has also long been the focus for security researchers, one of which belongs to the information leakage issue. On Android, inter-process communication (IPC) is one of the major reasons that cause information leak, when the app sends sensitive data to the recipient but without proper permission checking on the latter.

In particular, *intents* are frequently used for message passing within or across apps. An intent is a message that declares a recipient, optionally with data. It can be thought of as a self-contained object that specifies a remote procedure to call and includes the

associated arguments (remote procedure call, RPC). Intents are widely used for both inter-app communication and intra-app communication.

Intents are sent between three categories of Android components: Activities, Services, and Broadcast Receivers (excluding *Content Provider*). They can be used for different purposes, such as starting Activities, starting, stopping, and binding Services, broadcasting information to BroadcastReceivers, etc. All of these communications can be used with explicit or implicit Intents. Intents can be used for explicit or implicit communication. An explicit intent regulates that it should be delivered to a particular app with explicit app ID, whereas an implicit intent requests delivery to any app that satisfies certain properties. In other words, an explicit intent specifies the recipient by name, whereas an implicit intent's recipient(s) are eventually determined by the Android system based on the recipient's/recipients' support for certain operations. For example, consider an app that stores contact information. When a user clicks on one of his or her contacts' family address, the "contacts app" needs to ask another app to display a map of that location (in Android, it would be a ACTION_VIEW action with the data URI of the scheme "geo:0,0?q=my+street+address"). In this scenario, the contacts app could send an *explicit intent* directly to Google Maps (in most cases, it is enough to call intent.setPackage("com.google.android.apps.maps") followed by a startActivity(intent)). Alternatively, it can send an *implicit intent* that would be delivered to any app that also provides map service such as Yahoo! Maps (e.g., only startActivity(intent) is called). Using an explicit intent guarantees that it is delivered to the intended recipient, whereas implicit Intents allow for late runtime binding between different apps. Typically, the resolution result is checked by calling of intent.resolveActivity(getPackageManager()), which is unknown until runtime.

Chapter 3

HAWKEYE: Directed Greybox Fuzzing

3.1 Introduction and Motivation

Generally speaking, existing greybox fuzzers aim to cover as many program paths as possible within a limited time budget. However, there exist several testing scenarios where the programmers only care about certain specific paths or locations and hope these can be sufficiently tested. For example, if *MJS* [45] (a JavaScript engine for embedded devices) has a vulnerability discovered on MSP432 ARM platform, similar vulnerabilities may occur in the corresponding code for the other platforms. In such a situation, the fuzzer should be *directed* to reproduce the bug at these locations. In another scenario, the programmers need to check whether a given patch indeed fixes the bug(s) in the previous versions [46–49]. This requires the fuzzer to concentrate on those locations that are affected by patched code. In both scenarios, the fuzzer is expected to be directed to reach certain user specified locations in the target program. For clarity, we name such locations as *target locations*. Following the definition in AFLGo [16], we name those fuzzers that can fulfill the directed fuzzing task as *directed fuzzers*.

AFLGo [16] is a variant of the state-of-the-art fuzzer AFL. It is a directed greybox fuzzer (DGF) that reduces the reachability of target locations to an *optimization problem*. It adopts a meta-heuristic to promote the seeds with shorter distances. The distance is calculated based on the average the *weight* of basicblocks on seed's execution trace to the target basicblock(s), where the weight is determined by the edges in

the call graph and control flow graphs of the program, and the applied meta-heuristic is simulated annealing [50]. Owing to these, AFLGo improves the *power scheduling* for fuzzing directedness — how many new seeds (a.k.a. "energy" in AFLGo) should be generated from the current seed.

A pure dynamic execution can only get the feedback based on the traces it has *already* covered without any awareness about the predefined target locations. Thus, ***static analysis*** is required to extract the necessary information for guiding the execution towards the target locations for DGFs. The most widely adopted is to calculate the *distance* to the target locations for the components (e.g., basicblocks, functions) of the target program, so that when executed, DGFs can decide the *affinity* between current seed and the target locations from the components in the execution traces. The major challenge is that the distance needs to be efficiently calculated while not compromise certain desired features. In particular, it should help to keep to some extent the seed diversity [51]. For example, the existing seed distance calculation algorithm used in AFLGo always favors shortest path that leads to the target locations (see § 3.2.1), which may starve seeds that could be more easily mutated to reach the target location and further trigger crashes. The author of libFuzzer [28] argues that not taking into account *all the possible traces* may fail to expose the bugs hidden deeply in longer paths [52]. This derives the first desired property **P1**. Another challenge is that the static analysis should provide precise information with acceptable overheads. This is because that coarse static analyses will not benefit the dynamic fuzzing much, while *heavyweight static analyses themselves may take considerable time* before the dynamic fuzzing starts. This challenge derives the second desired property **P2**. Therefore, the first issue is *how a DGF applies proper static analysis effectively and efficiently to collect necessary metrics*.

After static analysis, there are several challenges in ***dynamic analysis*** — how to dynamically adjust different strategies to reach the target locations as fast as possible. The first challenge is how to properly *allocate energy* to the seeds with different distances and how to *prioritize* the seeds closer to the target locations. This derives the third desired property **P3**. The second challenge is how to adaptively change the mutation strategies, since GBFs may possess various mutation operators at both coarse-grained

(e.g., chunk replacement) and fine-grained (e.g., bitwise/bytewise flip) levels. This derives the fourth desired property **P4**. Therefore, the second issue is *how a DGF applies adaptive strategies during the dynamic fuzzing procedure*.

To emphasize the two aforementioned issues, we suggest that an ideal DGF is expected to hold the following desired properties (§3.2.2):

P1 It should provide a *robust* mechanism guiding the directed fuzzing by considering all the possible traces to the target locations.

P2 It should strike a *balance* between utilities and overheads as to static analysis.

P3 It should prioritize and schedule the seeds to reach target locations *rapidly*.

P4 It should adopt an *adaptive* mutation strategy across different program states.

We propose our solutions to achieve the 4 properties for DGF. For **P1**, we propose to apply the static analysis results to augment the adjacent-function distance (§3.4.2); and the function level distance and basicblock level distance should be calculated based on the augmented adjacent-function distance to simulate the affinities between functions (§3.4.3). Meanwhile, during fuzzing, we calculate *basicblock trace distance* and *covered function similarity* of the execution trace to that of the target functions (§3.4.4) by integrating the static analysis results with the runtime execution information. For **P2**, we propose to apply the analysis based on call graph (CG) and control flow graph (CFG), i.e., the function level reachability analysis, the points-to analysis for function pointers (indirect calls), and the basicblock metrics (§3.4.1). For **P3**, we propose to combine the basicblock trace distance and covered function similarity for solving the power scheduling problem (§3.4.4) and the seed priorization problem (§3.4.6). For **P4**, we propose to apply an adaptive mutation strategy according to the reachability analysis and covered function similarity (§3.4.5).

By taking these properties into account, we implemented our DGF, HAWKEYE, and conducted a thorough evaluation with various real-world programs. The experimental results show that in most cases, HAWKEYE outperforms the state-of-the-art greybox fuzzers in terms of the time to reach the target locations and the time to expose the

crashes. Particularly, HAWKEYE can expose certain crashes up to 7 times faster than the state-of-the-art AFLGo, reducing the time to exposure from 3.5 hours to 0.5 hours.

In practice, HAWKEYE has been successfully discovering crashes with the suspicious target locations reported by other vulnerability detection tools and successfully found more than 41 previously unknown crashes in projects Oniguruma [53], MJS [45], etc. All these vulnerabilities have been confirmed and fixed; among them, 15 vulnerabilities have been assigned unique CVE IDs.

The main contributions of this chapter are summarized as follows:

1) We analyzed the challenges in directed greybox fuzzing and summarized the four desired properties for DGFs.
2) We provided a measure of power function that can guide the fuzzer towards the target locations effectively.
3) We proposed a novel approach to boost the convergence speed to the target locations by utilizing power scheduling, adaptive mutation, and seed prioritization.
4) We implemented HAWKEYE that organically combines these ideas and thoroughly evaluated our results in crash reproduction and target location covering.

3.2 Desired Properties of DGF

In this section, we first show an example to illustrate the difficulties in DGF. Based on the observations from the example, we then propose four desired properties for an ideal DGF. Finally, we review the state-of-the-art DGF, namely AFLGo [16], with respect to these four desired properties.

3.2.1 Motivating Example

Figure 3.1 shows two execution traces related to CVE-2017-15939 [54], which is a NULL pointer dereference bug caused by an incomplete fix in CVE-2017-15023 [55]. This vulnerability is difficult for the existing GBFs to discover. For instance, AFL [27]

Functions in a Crashing Trace	**File & Line**	**Symbol**
`main`	`nm.c`:1794	M
...	...	...
`_bfd_dwarf2_find_nearest_line`	`dwarf2.c`:4798	a
`comp_unit_find_line`	`dwarf2.c`:3686	b
`comp_unit_maybe_decode_line_info`	`dwarf2.c`:3651	c
`decode_line_info`	`dwarf2.c`:2265	d
`concat_filename`	`dwarf2.c`:1601	T
...	...	Z
Functions in a Normal Trace	**File & Line**	**Symbol**
`main`	`nm.c`:1794	M
...	...	...
`_bfd_dwarf2_find_nearest_line`	`dwarf2.c`:4798	a
`scan_unit_for_symbols`	`dwarf2.c`:3211	e
`concat_filename`	`dwarf2.c`:1601	T
...	...	Z

FIGURE 3.1: Two execution traces relevant to CVE-2017-15939, where M is the entry function, T is the target function, and Z is the exit.

fails to detect this vulnerability within 24 hours in all the 10 different runs we conducted. This bug is triggered in `nm` from GNU binutils. In function `concat_filename`, a NULL pointer is assigned and used without checking, which triggers the segmentation fault. From a patch testing perspective, we would like to target `concat_filename` (subsequently, we will denote this as T) and guide the fuzzing to reproduce the crashing trace (i.e., $\langle a, b, c, d, T, Z\rangle$ in Figure 3.2).

For simplicity, in Figure 3.2, we illustrate only three representative traces for the CVE-2017-15939 by omitting 1) the overlapping functions before a and 2) the other traces that do not pass the target function T. The difficulty in discovering this CVE for the general GBFs (e.g., AFL) arises from the fact that the target function T is deeply hidden in the crashing trace. As shown in Figure 3.2, the call chain of $\langle a, e, T, Z\rangle$ is shorter than $\langle a, b, c, d, T, Z\rangle$.

Since most of the GBFs (such as AFL, LibFuzzer, etc.) are supposed to be *coverage oriented* and do not care particularly about the target locations, they may not put most of their efforts in generating seeds that reach function T and testing the function thoroughly. For DGFs, although there suppose to be some efforts to guide the fuzzing procedure to favor *some* traces leading to T and focus more on these traces, they may frequently miss *all* the traces. For example, if AFLGo detects that two traces

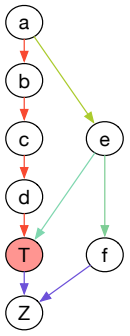

FIGURE 3.2: Fuzzing traces for Figure 3.1: $\langle a, b, c, d, T, Z\rangle$ is a crashing trace passing T, $\langle a, e, T, Z\rangle$ is a normal trace passing T; $\langle a, e, f, Z\rangle$ does not pass T.

can reach the target locations, it will likely favor the trace with a shorter path: the distance between the seed to the target is determined by the average distance of the components (basicblocks or functions) in the execution trace to the target locations, where the components' distance to the target locations is essentially determined by the number of edges between the components and the target locations. This mechanism causes AFLGo to give more energy to the trace $\langle a, e, T, Z\rangle$ since it reaches the target T *and* the induced trace distance is smaller than $\langle a, b, c, d, T, Z\rangle$; on the other hand, less attention is put on $\langle a, b, c, d, T, Z\rangle$, which however causes the crash under some circumstances. Worse still, other traces like $\langle a, e, f, Z\rangle$ may be mistakenly assigned with more energy. As a result, AFLGo was also not able to reproduce the crash within 24 hours in any of the 10 runs we conducted.

The challenges of the existing DGF roots in the following aspects: 1) the target functions may appear in several places in target program, and multiple different traces may lead to the target. 2) since the call graph majorly affects the calculation of the trace distance (the dissimilarity with the target locations), it needs to be accurately built; in particular, the indirect calls among functions should not be ignored. If the above two issues are not well handled, the distance-based guiding mechanism for DGF will get hindered and fail in such cases.

3.2.2 Desired Properties of Directed Fuzzing

We observe that an ideal DGF should possess the following desired properties.

P1. DGF should define a *robust* distance mechanism to guide directed fuzzing by avoiding the bias to certain traces Different from general GBFs, to reach the specified locations, there may exist several execution traces towards them. More often than not, a target function could appear several times in the code and be called even from different entries of the code. Without any static information as guidance, during the fuzzing process, the fuzzer knows nothing about the execution traces that cover the target locations before they have been executed; and even if the locations have already been covered, the fuzzer does not know whether *other* traces can lead to them. Taking Figure 3.2 as an example, in the AFL fuzzing process, trace $\langle a, b, c, d, T, Z\rangle$ may not be ever exercised by the existing seeds due to the existence of a strong precondition before a. Therefore, *the guiding mechanism should provide the knowledge of all the possible traces that lead to the target locations and guide the mutation towards it via gradually reducing the distance*.

However, for DGF, awareness of all the possible traces towards target locations is not enough: the distance to specified locations for all traces should be properly calculated so that all traces reachable to them will be assigned more energy compared to other traces. For Figure 3.2, T is the target and we would like to check the functionality of T. Note that if we know in advance that *only* the traces that involve d and T may cause crashes, we can set *both* d and T as target locations. Intuitively, traces $\langle a, e, T, Z\rangle$ and $\langle a, b, c, d, T, Z\rangle$ should be treated without bias as both of them can lead to the target location, while $\langle a, e, f, Z\rangle$ should be less important as it misses the target location.

P2. DGF should strike a *balance* between efficiency and effectiveness for static analysis. Effective static analysis can benefit the dynamic fuzzing procedure in two aspects: 1) In real-world C/C++ programs, there are indirect function calls (e.g., passing a function pointer as a parameter in C, or using function objects and pointers to member functions in C++). In the presence of indirect calls, call sites cannot be observed directly from the source code or binary instructions. So the trade-offs between overheads and utilities need to be made for analyzing them. 2) Not all call relations should be treated equally. For example, some functions occur multiple times in its calling functions, implying that they have a higher chance to be called at runtime. From the static analysis perspective, we need to provide a way to distinguish these scenarios. As to function

level distances between functions that have *immediate* calling relations, it is intuitive that callees called in multiple times in different branches should be "closer" to the caller.

To sum up, taking Figure 3.2 as an example, *the desired design* for the DGF is: 1) if function a (transitively) calls T indirectly (i.e., one or more calls in the chain $a \rightarrow b \rightarrow c \rightarrow d \rightarrow T$ are through function pointers), the static analysis should capture such indirect calls, otherwise the distance from a to T will be not available (i.e., treated as unreachable). 2) if the callee appears in more different branches and occurs more times in its caller, a smaller distance should be given since it may have more chance of being called for reaching the target(s). However, modeling the *actual* branch conditions in the static phrase is impractical due to the inherent limitations of static analysis. For example, given a nontrivial code segment, it is hard to predict whether the true branch of a predicate will be executed more often than its false branch during runtime. On the other hand, tracking symbolic conditions *dynamically* would be too time costly in a greybox fuzzing setting.

P3. DGF should *prioritize* seeds close to target locations. AFL determines how many new seeds should be generated (i.e., "energy") from a seed to improve the fuzzing effectiveness (i.e., increase the coverage); this is termed "power scheduling" in [15, 16]. In directed fuzzing, the goal of fuzzing is not to reach *the upper-limit of coverage* as fast as possible but reach the *particular target locations* as fast as possible. Hence, power scheduling in DGF should determine how many new seeds should be generated from a seed in order to get a new seed that leads to the target locations [16]. Similarly, the seed prioritization in DGF is to determine an optimized fuzzing order of seeds to reach target locations as fast as possible. Both of them can be guided by the distance-based mechanism which measures the affinity between the current seed to the target locations.

For *power scheduling*, the desired design is that the seed trace with a smaller distance to the target locations should be assigned more energy for fuzzing, as the trace closer to the target locations gets a better chance to reach there. Therefore, $\langle a, e, T, Z\rangle$ should be allocated with similar energy with $\langle a, b, c, d, T, Z\rangle$, and $\langle a, e, f, Z\rangle$ should have less energy than the previous two. For *seed prioritization*, seeds that have a smaller distance ("closer") to the target locations should be fuzzed earlier in subsequent mutations. Therefore, $\langle a, e, T, Z\rangle$ and $\langle a, b, c, d, T, Z\rangle$ should be put ahead of $\langle a, e, f, Z\rangle$.

P4. DGF should adopt an *adaptive* mutation strategy at different program states. GBFs usually apply different mutations, such as bitwise flip, byte rewrite, chunk replacement, to generate new seeds from the existing one. In general, these mutators can be categorized into two levels: fine-grained mutations (e.g., bitwise flip) and coarse-grained mutations (e.g., chunk replacement). Although there is no direct evidence that fine-grained mutations will likely preserve the execution traces, it is widely accepted that a coarse-grained random mutation has a high chance to greatly change the execution trace. Therefore, the desired design is that when a seed has already reached the target locations (including target lines, basicblocks or functions), it should be given fewer chances for coarse-grained mutations.

For the example in Figure 3.2, consider the case where the DGF has already reached the target function via trace $\langle a, b, c, d, T, Z\rangle$, but the crash is not triggered yet. Now, the DGF should allocate fewer chances for coarse-grained mutations for the input of $\langle a, b, c, d, T, Z\rangle$. Meanwhile, if DGF has just started up and $\langle a, b, c, d, T, Z\rangle$ has not been reached yet, the DGF should give more chances for coarse-grained mutations.

3.2.3 AFLGo's Solution

In this section, we evaluate the solution of AFLGo against the four desired properties to demonstrate the significances of these properties in DGF.

FIGURE 3.3: Approach Overview of HAWKEYE

For P1. In AFLGo, the seed distance to a specific target basicblock/function is calculated as the average shortest distance of basicblocks/function the seed covers. For example, in Figure 3.2, where the nodes are functions, for trace $a \to b \to c \to d \to T \to Z$, given the fact that $d(a, T) = 2$ (shortest path: $a \to e \to T$), $d(b, T) = 3$ (shortest path: $b \to c \to d \to T$), $d(c, T) = 2$ (shortest path: $c \to d \to T$), $d(d, T) = 1$ (shortest path: $d \to T$), we can see $d_s(\langle a, b, c, d, T, Z\rangle) = (2 + 3 + 2 + 1 + 0)/5 = 1.6$. Analogously, $d_s(\langle a, e, T, Z\rangle) = (2 + 1 + 0)/3 = 1$ and $d_s(\langle a, e, f, Z\rangle) = (2 + 1)/2 = 1.5$ [1]. Given these three execution traces, the energy assigned to them will be $\langle a, e, T, Z\rangle$

[1] In practice, AFLGo calculates the trace distance at the *basicblock* level with harmonic mean of the accumulative distance; nevertheless, the essential idea is the same.

$> \langle a, e, f, Z\rangle > \langle a, b, c, d, T, Z\rangle$. This is problematic: the normal trace $\langle a, e, T, Z\rangle$ is overemphasized; the crashing trace $\langle a, b, c, d, T, Z\rangle$ is however considered the least important, even less important than the trace $\langle a, e, f, Z\rangle$ that fails to reach the target T.

For P2. AFLGo only considers the explicit call graph. Due to this, all function pointers are treated as *external nodes* which are ignored during distance calculation. This means that, in an extreme case, if the target function is called via a function pointer, its distance from the actual caller is undefined. For example, in Figure 3.2, if d and e call T via function pointers, both d and e will be mistakenly considered unreachable to T; consequently, all nodes except for T will be considered unreachable to T. Therefore essentially there is no directedness in such a case. Besides, AFLGo counts the same callee in its callers only once and does not differentiate various call patterns between the caller and callee (see §3.4.2). The function level distance is calculated on the call graph with the Dijkstra shortest path, assuming the weight of two adjacent nodes (functions) in the call graph always to be 1, which will distort the distance calculation.

For P3. AFLGo applies a simulated annealing based power scheduler: it favors those seeds that are closer to the target locations by assigning more energy to them for mutation; the applied cooling schedule initially assigns smaller weight on the effect of "distance guidance", until it reaches the "exploitation" phrase. It solves the "exploration vs exploitation" problem [56] and mitigates the imprecision issue brought by the statically calculated basicblock level distance. In our opinion, this is an effective strategy. The problem is that there lacks a prioritization procedure so the newly generated seeds with a smaller distance may wait for a long to be mutated.

For P4. The mutation of AFLGo comes from AFL's two non-deterministic strategies: 1) *havoc*, which does purely random mutations such as bit flips, chunk replacement, etc; 2) *splice*, which generates seeds from some random byte parts of two existing seeds. Notably, during runtime AFLGo excludes all the deterministic mutation procedures and relies purely on the power scheduling on havoc/splice strategies. The randomness of these two strategies can indeed favor those with smaller distances to the target locations. However, it may also destroy the existing seeds that are close to them. In fact, some subtle vulnerabilities can only be reached with some special preconditions.

In reality, an incomplete fix may still leave some concern cases to be vulnerable; for example, CVE-2017-15939 is caused by an incomplete fix for CVE-2017-15023. Hence, AFLGo lacks the adaptive mutation strategies, which will mutate arbitrarily even when the current seeds are close to the specified locations enough.

Summary. We summarize the following suggestions to improve DGFs:

(1) For **P1**, an accurate distance definition is needed to retain trace diversity and mitigate the bias on short traces.

(2) For **P2**, both direct and indirect calls need to be analyzed; different call patterns need to be distinguished during static distance calculation.

(3) For **P3**, a moderation to the current power scheduling and a distance-guided seed prioritization strategy are preferable.

(4) For **P4**, an adaptive mutation strategy is needed to optimally apply fine-grained and coarse-grained mutations.

3.3 Approach Overview

In this section, we briefly introduce the workflow of our proposed approach, named HAWKEYE. An overview of HAWKEYE is given in Figure 3.3, which consists of two major components, i.e., *static analysis* and *fuzzing loop*.

3.3.1 Static Analysis

The inputs of static analysis are the *program source code* and the *target locations* (i.e., the lines of code that the fuzzer is directed to reach). We derive the basicblocks and functions where the target locations reside in, and call them *target basicblocks* and *target functions*, respectively. The main output of the static analysis is the *instrumented program binary* with the information of *basicblock level distance*.

Firstly, we precisely construct the call graph (CG) of the target program based on the inclusion-based pointer analysis [57] to include all the possible calls. For each function, we construct its control flow graph (CFG) (§3.4.1).

Secondly, we compute several utilities to facilitate the directedness in HAWKEYE based on CG and CFG (§3.4.3).

(1) **Function level distance** is computed based on CG by augmenting adjacent-function distance (§3.4.2). This distance is utilized to calculate the *basicblock level distance*. It is also used during the fuzzing loop to calculate the *covered function similarity* (§3.4.4).

(2) **Basic block level distance** is computed based on the function level distance with the CG and the functions' CFGs. This distance is statically instrumented for each basicblock considered to be able to reach one of the target locations. In the fuzzing loop, it is also used to calculate the *basicblock trace distance* (§3.4.4).

(3) **Target function trace closure** is computed for each target location according to the CG to obtain the functions that can reach the target locations. It is used during the fuzzing loop to calculate the *covered function similarity* (§3.4.4).

Lastly, the target program is instrumented to keep track of the edge transitions (similar to AFL), the accumulated basicblock trace distance, and the covered functions.

3.3.2 Dynamic Fuzzing Loop

The inputs of fuzzing loop are the *instrumented program binary*, the *initial seeds*, the *target locations* as well as the information of *function level distance* and *target function trace closure*. The outputs of fuzzing loop are the seeds that cause abnormal program behaviors such as crashes or timeouts.

During fuzzing, the fuzzer selects a seed from a priority seed queue. The fuzzer applies a *power scheduling* against the seed with the goal of giving those seeds that are considered to be "closer" to the target locations more mutation chances (a.k.a. energy,

§3.4.4). Specifically, this is achieved through a power function, which is a combination of the *covered function similarity* and the *basicblock trace distance*. For each newly generated seed during mutation, after capturing its execution trace, the fuzzer will calculate the covered function similarity and the basicblock trace distance based on the utilities (§3.3.1). For each input execution trace, its basicblock trace distance is calculated as the accumulated basicblock level distances divided by the total number of executed basicblocks; and its covered function similarity is calculated based on the intersection of covered functions and the target function trace closure, as well as the function level distance.

After the energy is determined, the fuzzer adaptively allocates mutation budgets on two categories of mutations according to mutators' granularities on the seed (§3.4.5). Afterwards, the fuzzer evaluates the newly generated seeds to prioritize those that have more energy or that have reached the target functions (§3.4.6).

3.4 Methodology

This section elaborates the key components in Figure 3.3 featuring the four properties.

3.4.1 Graph Construction

To calculate the accurate distance from a seed to the target locations, we first build up the CG and CFG, then combine them to construct the final inter-procedural CFG. Note that CG is used to compute the function level distance in §3.4.2 and §3.4.3, CFG together with CG (i.e., inter-procedural CFG) is used to compute the basicblock distance in §3.4.3.

To identify the indirect call in call graph, we propose to apply the inclusion-based pointer analysis [57] against the function pointers of the whole program. The core idea of this algorithm is to translate the input program with statements of the form $p := q$ to constraints of the form "q's points-to set is a subset of p's points-to set". Essentially, the propagation of the points-to set is applied with four rules namely *address-of*, *copy*,

assign, *dereference*. This analysis is context-insensitive and flow-insensitive, meaning that it ignores both the calling context of the analyzed functions and the statement ordering inside functions, and eventually only computes a single points-to solution that holds for all the program points. Usually, a fixed point of the points-to sets will be reached at the end of the analysis. Among these, points-to sets of the function pointers inside the whole program are calculated, resulting in a relatively precise call graph including all the possible direct and indirect calls. The complexity of this pointer analysis is $\Theta(n^3)$. The reason that we do not apply context-sensitive or flow-sensitive analyses lies in the fact that they are computationally costly and not scalable to large projects. Despite that, our call graph is still much more precise than the one generated by LLVM's builtin APIs, which does not contain any explicit nodes that represent indirect calls.

The control flow graph of each function is generated based on LLVM's IR. The inter-procedure flow graph is constructed by collecting the call sites in all the CFGs and the CG of the whole program. By applying these static analyses, we achieve **P2**.

3.4.2 Adjacent Function Distance Augmentation

To achieve **P1**, we propose to implement a lightweight static analysis that considers the patterns of the (immediate) call relation based on the generated call graph. As discussed in **P2**, under different context, the distances from the calling function to the immediately called function may not be exactly the same. Given functions f_a, f_b, f_c, there may exist several different call patterns in the call graph. For example, in Figure 3.4a and Figure 3.4b, there are calls $f_a \rightarrow f_b$ and $f_a \rightarrow f_c$ in both cases. However, in Figure 3.4a f_a is bound to call f_b (since f_b appears in both *if* and *else* branches in f_a), but not necessary to call f_c; in Figure 3.4b, both f_b and f_c are not necessary to be called by f_a. From a probability perspective, we would think that in both cases the distance from f_a to f_b should be smaller than the distance from f_a to f_c, and the distance from f_a to f_b in Figure 3.4a should be smaller than that in Figure 3.4b.

Therefore, we propose two metrics to augment the distance that is defined by immediate calling relation between caller and callee.

```
1 void fa(int i) {
2   if (i > 0) {
3     fb(i);
4   } else {
5     fb(i*2);
6     fc();
7   }
8 }
```

(a)

```
1 void fa(int i) {
2   if (i > 0) {
3     fb(i);
4     fb(i*2);
5   } else {
6     fc();
7   }
8 }
```

(b)

FIGURE 3.4: An example that illustrates different call patterns.

(1) Call site occurrences $C_N(f_1, f_2)$ of a certain callee f_2 for a given caller f_1. More occurrences of callee could incur more chance that callee will be dynamically executed with more different (actual) parameters, and in return the distance between the caller to the callee will be smaller. We apply a factor $\Phi(f_1, f_2) = \frac{\phi \cdot C_N(f_1,f_2)+1}{\phi \cdot C_N(f_1,f_2)}$ to denote this effect, where ϕ is a constant value.

(2) The number of basicblocks $C_B(f_1, f_2)$ in the caller that contains at least one call site of the callee. The rationale is that, with more branches that have a call site, more different execution traces will include the callee. The factor function $\Psi(f_1, f_2) = \frac{\psi \cdot C_B(f_1,f_2)+1}{\psi \cdot C_B(f_1,f_2)}$ denotes this effect, and ψ is a constant value.

By default, we choose $\phi = 2$ and $\psi = 2$, to balance the effects of the considered two metrics. Note that both factor functions are monotone decreasing functions; also, $\Phi(f_1, f_2)$ converges to 1 when $C_N(f_1, f_2) \to \infty$ and $\Psi(f_1, f_2)$ converges to 1 when $C_B(f_1, f_2) \to 1$. Given a (direct or indirect) immediate function call pair (f_1, f_2) where f_1 is the caller and f_2 is the callee, the original distance between f_1 and f_2 is 1. Now, with the two metrics mentioned above, we can define the augmented distance between the function pairs that holds an immediate call relation. The final adjustment factor will be a multiplication of Φ and Ψ, and the *augmented adjacent-function distance* is

$$d'_f(f_1, f_2) = \Psi(f_1, f_2) \cdot \Phi(f_1, f_2) \tag{3.1}$$

where $d'_f(f_1, f_2)$ refers to the *augmented direct* function distance.

As an example, in Figure 3.4a, for f_b, $C_N(f_a, f_b) = 2$, $C_B(f_a, f_b) = 2$; and for f_c, $C_N(f_a, f_c) = 1$, $C_B(f_a, f_c) = 1$. Assume $\phi = 2$ and $\psi = 2$ and assume the original distance $d_f(f_a, f_b) = d_f(f_a, f_c) = 1$, the augmented distances will be $d'_f(f_a, f_c) = \frac{3}{2} \cdot \frac{3}{2} = 2.25$, and $d'_f(f_a, f_b) = \frac{5}{4} \cdot \frac{5}{4} = 1.56$.

A special case not shown in the above examples is that some branches form cycles (i.e., loops). Indeed, these functions may be called multiple times at runtime. However, it is uncertain the sense that *how many times* they will be executed across different runs when fed with different seeds. Fortunately, actual execution on one call site of a callee inside one loop typically has a similar effect — the loop explores the similar program states and benefits less in covering new paths. Hence, the function call inside the loop does not bring many execution trace diversities like the scenario where the same callee occurs in multiples branches with significantly different parameters. In the future, we plan to apply loop summarizations based on invariants [58–63] to specialize these cases.

The applied approach aims to make a trade-off between the efficiency and the utility of the static analysis. Therefore, we do not consider the solution space of any branch condition that may affect the *runtime reachability* in the CFG. For example, in Figure 3.4a, if we change the condition check $i > 0$ to be $i == 0$, the true branch will be executed only when the input value of i is 0. It is tempting to assign a smaller distance to the code segments in the false branch. However, since the target program is usually nontrivial, it is impractical to statically formulate the exact constraint set of the preconditions before reaching function f_a and predicate the branches' actual execution probabilities. One common scenario is that the branch condition *i==0* is used for checking the return status code of an external function call, at runtime it may actually execute the true branch more often than the false branch.

3.4.3 Directedness Utility Computation

In §3.4.2, the augmented function distance is calculated on two adjacent functions according to their call patterns. By assigning the adjacent-function distance as the weight of the edges in the call graph, we can calculate the function level distance for any two functions with the Dijkstra shortest path algorithm, beyond which we can further derive the basicblock level distance. Besides, we also compute the target function trace closure which will be used to calculate the covered function similarity in §3.4.4.

Function Level Distance. This distance is calculated according to CG. It tells the (average) distance from the current function to target functions. Given a function n, its

distance to the target function set T_f is defined as:

$$d_f(n, T_f) = \begin{cases} \textbf{undefined.} & \text{if } R(n, T_f) = \emptyset \\ [\Sigma_{t_f \in R(n,T_f)} d_f(n, t_f)^{-1}]^{-1} & \text{otherwise} \end{cases} \tag{3.2}$$

where $R(n, T_f) = \{t_f | \textbf{reachable}(n, t_f)\}$, which is the set of target functions that can be statically reached from n in CG, and $d_f(n, t_f)$ is the *dijkstra shortest path based on augmented function distance* from n to a given target function t_f in CG.

Basic Block Level Distance. Given a function n and two basicblocks m_1 and m_2 inside, the basicblock level distance $d_b^o(m_1, m_2)$ is defined as the minimal number of edges from m_1 to m_2 in the CFG $G(n)$. The set of functions called inside basicblock m is denoted as $C_f(m)$, then $C_f^T(m) = \{n | R(n, T_f) \neq \emptyset, n \in C_f(m)\}$, and $Trans_b = \{m | \exists G(n), m \in G(n), n \in F, C_f^T(m) \neq \emptyset\}$, where F is the set of all functions. Given a basicblock m, its distance to the target basicblocks T_b are defined as:

$$d_b(m, T_b) = \begin{cases} 0 & \textbf{if } m \in T_b \\ c \cdot \min_{n \in C_f^T(m)}(d_f(n, T_f)) & \textbf{if } m \in Trans_b \\ [\Sigma_{t \in Trans_b}(d_b^o(m, t) + d_b(t, T_b))^{-1}]^{-1} & otherwise \end{cases} \tag{3.3}$$

where c is a constant that magnifies function level distance.

Note that Equation 3.2 and 3.3, on their own, are the same as those in AFLGo [16]. However, $d_f(n, t_f)$ for these equations in AFLGo is simply the Dijkstra shortest distance on a CG where the weight of edges (i.e., adjacent function distance) is 1.

Target Function Trace Closure. This utility, $\xi_f(T_f)$, is calculated by collecting all the predecessors that can statically lead to the target functions T_f, until the entry function *main* has been reached. We choose *not* to exclude those that are *considered* unreachable from entry function due to the limitations of static analysis. In the example in Figure 3.2, $\xi_f(T_f) = \{a, b, c, d, e, T\}$.

3.4.4 Power Scheduling

During fuzzing, we apply power scheduling on a selected seed based on two dynamically computed metrics: basicblock trace distance and target function trace similarity.

Basic Block Trace Distance. The distance between seed s to the target basicblocks T_b is defined as:

$$d_s(s, T_b) = \frac{\Sigma_{m \in \xi_b(s)} d_b(m, T_b)}{|\xi_b(s)|} \tag{3.4}$$

where $\xi_b(s)$ is the execution trace of a seed s and contains all the basicblocks that are executed. Hence, the basic idea of Equation 3.4 is that: for all the basicblocks in the execution trace of s, we calculate the average basicblock level distance to the target basicblocks T_b. Note that Equation 3.4 is also the same as the one in AFLGo [16].

It then applies a feature scaling normalization to get the final distance $\tilde{d}_s(s, T_b) = \frac{d_s(s,T_b) - minD}{maxD - minD}$ where $minD$ (or $maxD$) is the smallest (or largest) distances ever met.

Covered Function Similarity. This metric measures the similarity between a seed's execution trace and the target execution trace *at function level*. We do not track basicblock level trace similarity since that would introduce considerable overheads. The similarity is calculated based on the intuition that seeds covering more functions in the expected traces have more chances to be mutated to reach the target locations. This similarity is calculated by tracking the seed's covered function *sets* (denoted as $\xi_f(s)$) and comparing it with the target function trace closure $\xi_f(T_f)$. In the example in Figure 3.2, $\xi_f(\langle a, b, c, d, T, Z\rangle) = \{a, b, c, d, T\}$, $\xi_f(\langle a, e, T, Z\rangle) = \{a, e, T\}$ and $\xi_f(aefZ) = \{a, e\}$.

The covered function similarity is then determined by the following formula:

$$c_s(s, T_f) = \frac{\Sigma_{f \in \xi_f(s) \cap \xi_f(T_f)} d_f(f, T_f)^{-1}}{|\xi_f(s) \cup \xi_f(T_f)|} \tag{3.5}$$

$d_f(f, T_f)$ is the function level distance calculated with Equation 3.2. Similar to d_s, a feature scaling normalization is also applied and the final similarity is denoted as $\tilde{c}_s$. Note that this similarity metric is uniquely proposed in our approach.

Scheduling. Scheduling deals with the problem how many mutation chances will be assigned to the given seed. The intuition is that if the trace that the current seed executes is "closer" to any of the expected traces that can reach the target location in the program, more mutations on that seed should be more beneficial for generating expected seeds. A scheduling purely based on *trace distance* may favor certain patterns of traces. For AFLGo, as mentioned in **P2**, the shorter paths will be assigned more energy, which may starve longer paths that are still reachable to the target locations. To *mitigate* this,

we propose the *power function* that considers both *trace distance* (based on basicblock level distance) and *trace similarities* (based on covered function similarity):

$$p(s, T_b) = \tilde{c}_s(s, T_f) \cdot (1 - \tilde{d}_s(s, T_b)) \tag{3.6}$$

Obviously, the value of $p(s, T_b)$ fits into $[0, 1]$ since both the multipliers are in $[0, 1]$.

Compared to AFLGo's approach, which only considers basicblock trace distance (d_s, or $\tilde{d}_s$), our power function balances the effect of shorter paths and the longer paths that can reach the target. Logically, there are some differences between c_s and d_s:

(1) d_s considers both the effects of CG and the CFGs; c_s considers only the effects of CG. For d_s, the major effect is still CG due to the magnification factor c used in Equation 3.2.

(2) d_s does not penalize traces that do not lead to the target locations, while c_s penalizes them via a union of $\xi_f(s)$ (by tracking function level traces) and $\xi_f(T_f)$.

(3) Given multiple traces that can lead to the target locations, d_s favors those that have shorter lengths, but c_s favors those with longer lengths of common functions in the expected trace.

In this sense, $p(s, T_b)$ strives a balance between shorter traces and longer traces that can reach the target locations with *two heterogeneous metrics*. Admittedly, there may still exist some bias. One of the scenarios is that the power function may assign more energy to a seed that covers many functions in the target function trace closure. For example, assume two traces that can reach the target function T: $\langle a, b, c, d, T, Z\rangle$ and $\langle a, e, f, g, T, Z\rangle$; and the target function trace closure is $\langle a, b, c, d, e, f, g, T\rangle$. The power function may assign much energy to a seed with trace $\langle a, b, c, d, e, f, g, Z\rangle$ which does not reach the target function T. This is *not* an issue in our opinion: since this seed has covered many "expected" functions, it has a high chance to be "close" to the target; with proper mutations, it is likely to be flipped to mutants that can indeed touch the target.

In HAWKEYE, the power function determines the number of mutation chances to be applied on the current seed (i.e., energy); it is also used during the seed prioritization to determine whether the *mutated* seeds should be favored.

Algorithm 2: $adaptiveMutate()$: Adaptive Mutation

input : s, the seed to be fuzzed after power scheduling
output: $\mathcal{M}_s$, the map to store the new mutated seed, whose key is the seed and whole value is the energy of the seed
const. : γ, the constant ratio to do fine-grained mutation
const. : δ, the constant ratio to be adjusted

$\mathcal{M}_s = \emptyset$;
$p \leftarrow s.getScore()$;
if $reachTarget(s) == false$ **then**
  $\mathcal{S}' \leftarrow coarseMutate(s, p * (1 - \gamma))$;
  for s' *in* $\mathcal{S}'$ **do**
    $\mathcal{M}_s \leftarrow \mathcal{M}_s \cup \{(s', s'.getScore())\}$
  $\mathcal{S}'' \leftarrow fineMutate(s, p * \gamma)$;
  for s'' *in* $\mathcal{S}''$ **do**
    $\mathcal{M}_s \leftarrow \mathcal{M}_s \cup \{(s'', s''.getScore())\}$
else
  $\mathcal{S}' \leftarrow coarseMutate(s, p * (1 - \gamma - \delta))$;
  for s' *in* $\mathcal{S}'$ **do**
    $\mathcal{M}_s \leftarrow \mathcal{M}_s \cup \{(s', s'.getScore())\}$
  $\mathcal{S}'' \leftarrow fineMutate(s, p * (\gamma + \delta))$;
  for s'' *in* $\mathcal{S}''$ **do**
    $\mathcal{M}_s \leftarrow \mathcal{M}_s \cup \{(s'', s''.getScore())\}$

3.4.5 Adaptive Mutation

In §3.4.4, for each seed, the output of power scheduling is the energy (a.k.a. the times of applied mutations), which will be the input of the step of our adaptive mutation. The problem is that, given the total energy available for a seed, we still need to assign the number of mutations for *each type of mutators*.

In general, two categories of mutators are used in GBFs. Some are coarse-grained in the sense that they change a bulk of bytes during the mutations. Others are fine-grained since they only involve a few byte-level modifications, insertions or deletions. For coarse-grained mutations, we consider them to be:

(1) **Mixed havoc**. It includes several chunk mutations, namely deleting a bulk of bytes, overwriting the given chunk with other bytes in the buffer, deleting certain lines, duplicating certain lines multiple times, etc. The actual mutation involves their combinations.

Algorithm 3: $coarseMutate()$: Coarse-Grained Mutation

input : s, the seed to be fuzzed after power scheduling
input : i, the number of iterations to do mutation on the seed
output: $\mathcal{S}$, the set to store the *new* mutated seed
const. : σ, the constant ratio to do semantic mutations
const. : ζ, the constant ratio to do mixed havoc mutations

```
1 S = ∅;
2 if needSemMutation(s) == true then
3     S ← S ∪ semMutate(s, i * σ) ;
4     S ← S ∪ coarseHavoc(s, i * (1 − σ) * ζ);
5     S ← S ∪ splice(s, i * (1 − σ) * (1 − ζ)) ;
6 else
7     S ← S ∪ coarseHavoc(s, i * ζ);
8     S ← S ∪ splice(s, i * (1 − ζ)) ;
```

(2) **Semantic mutation**. It is used when the target program is known to process semantic relevant input files such as javascript, xml, css, etc. In detail, this follows Skyfire [21], which includes three meta mutations, inserting another subtree into a random AST position, deleting a given AST, and replacing the given position with another AST.

(3) **Splice**. It includes a crossover between two seeds in the queue and subsequent mixed havocs.

Algorithm 2 shows the workflow of our adaptive mutation, given a seed s. The basic idea is to give less chance of coarse-grained mutations when the seed s can reach the target functions (at line 10 in Algorithm 2). Once the seed reaches the target locations, the times of doing fine-grained mutations increase from $p * \gamma$ (line 7) to $p * (\gamma + \delta)$ (line 14), but the times of doing coarse-grained mutation decrease from $p * (1 - \gamma)$ (at line 4) to $p * (1 - \gamma - \delta)$ (line 11). Here, $s.getScore()$ at line 2 is to get the energy assigned to the seed according to the power function value calculated in Equation 3.6.

$fineMutate()$ in Algorithm 2 simply applies a random fine-grained mutation (e.g., bit/byte flippings, arithmetics on some bytes) for the seed. Algorithm 3 shows the details for coarse-grained mutation strategy $coarseMutate()$. Given a seed s and the iteration times of mutations i, the basic idea is to apply semantic mutations (line 2) only when it is necessary (line 3). The constraints $needSemMutation(s)$ returns true if the following conditions are satisfied: 1) our fuzzer detects that the input file is a semantic-relevant

Algorithm 4: Seed Prioritization

input : s, the seed to be processed
output: $\mathcal{Q}_1$, the tier 1 queue to store the most important seeds
output: $\mathcal{Q}_2$, the tier 2 queue to store the important seeds
output: $\mathcal{Q}_3$, the tier 3 queue to store the least important seeds
const. : μ, the threshold of energy value for accepting important seeds

```
1  Q1 = Q2 = Q3 = ∅;
2  if seedIsNew(s) == true then
3      if seedWithNewEdge(s) == true then
4          Q1 ← Q1 ∪ {s};
5      else if s.powerEnergy() > μ then
6          Q1 ← Q1 ∪ {s};
7      else if reachTarget(s) == true then
8          Q1 ← Q1 ∪ {s};
9      else
10         Q2 ← Q2 ∪ {s};
11 else
12     Q3 ← Q3 ∪ {s};
```

input file such as javascript, xml, css, etc; 2) The previous semantic mutations have not failed. Otherwise (line 6), mixed havoc mutations will get more times ($i * \zeta$, at line 7) than the necessary case ($i * (1 - \sigma) * \zeta$, at line 4), meanwhile splice mutations will also get more times ($i * (1 - \zeta)$ line 8) than necessary ($i * (1 - \sigma) * (1 - \zeta)$, at line 5). In practice, we assign empirical values to: $\gamma = 0.1$, $\delta = 0.4$, $\sigma = 0.2$, $\zeta = 0.8$.

Note that all these new generated seeds in $\mathcal{M}_s$, together with the original seeds, will be put into the seed queue for future fuzzing. Actually, before fuzzing them, we will prioritize them to improve the efficiency of directed fuzzing (see §3.4.6).

3.4.6 Seed Prioritization

Not all the seeds have equal or similar priorities. Ideally, the queue that stores the seeds to be mutated should be a priority queue. However, the scoring may be biased (due to the limitations of static analyses, etc.), and the insertion operations on the priority queue take a complexity of $\Theta(\log n)$, which is costly since the queue can be quite long and the insertion operation can be frequent. Therefore it is not beneficial *in practice* [1]. Instead,

[1] In fact, the well-known AFL fuzzer only maintains a linked list with some probabilistic to skip seeds that do not cover new edges; the complexity of the insertion is $\Theta(1)$.

we provide a three-tiered queue which appends newly generated seeds into different categories according to their scores. Seeds in the top-tiered queue (tier 1) will be picked firstly, then the second-tiered (tier 2), and finally the lower-tiered (tier 3). This imitates a simplified priority queue with constant time complexity.

Algorithm 4 shows the seed prioritization strategy for a new seed mutated from the previous step of adaptive mutation. The basic idea is: we should prioritize the *newly generated* seeds that 1) cover new traces 2) have bigger similarity values with the target seeds (i.e., power function values) 3) cover the target functions. We favor seeds that cover new traces since we still have to explore *more execution paths* that have the potential to lead to the target locations; this is necessary when the initial seeds are far from the target locations. The other two prioritization strategies, i.e., comparing similarity values and checking whether the target function have been reached, are specific to directed fuzzing. Note that although these two strategies are relevant, neither of them can be deduced from the other. For the other newly generated seeds, they are put in the second-tiered queue. On the other hand, seeds that have (just) been mutated are assigned with the least priority. In practice, HAWKEYE also applies AFL's loop bucket approach (see [64]) to filter out a large number of "equivalent" seeds that bring no new coverage as to loop iterations. The prioritization strategies will be applied on the remaining seeds. Therefore, there will not be too many seeds filling up the top-tier queue.

By combining all the static and dynamic techniques mentioned above, for the CVE exampled in §3.2.1, HAWKEYE successfully reproduced the crash with a time budget of 24 hours in 3 out of the 10 runs we conducted when fed with the same initial seeds, which is a significant improvement on both AFL and AFLGo for this case.

3.5 Evaluation

We implemented our instrumentation on top of FOT's static instrumentation mode (c.f. §5) and the pointer analysis is based on the interprocedural static value-flow analysis tool called SVF [65]; this part takes about 2000 lines of C/C++ code based on the LLVM [33] infrastructure. For directed fuzzing purpose, we add another

TABLE 3.1: Program statistics for our tested programs.

Project	Program	Size	ics	cs	ics/cs	# of C_B>1	# of C_N>1	t_s
Binutils	cxxfilt	2.8M	3232	12117	26.67%	8813	8879	735s
Oniguruma	testcu	1.3M	556	2065	26.93%	3037	3101	5s
mjs	mjs	277K	130	3277	3.97%	309	334	3s
libjpeg	libjpeg	810K	749	1827	41.00%	144	152	2s
libpng	libpng	228K	449	1018	44.11%	61	61	2s
freetype2	freetype	1.6M	627	5681	11.30%	6784	7117	4s

4000 lines of code for tracing functions, calculating power function for seeds, distinguishing graininess of mutators, and so on. More details are available on https://sites.google.com/view/ccs2018-fuzz.

For each program with the given target locations, the instrumentation of HAWKEYE consists of three parts: 1) basicblock IDs that track the execution traces 2) basicblock distance information that determines basicblock trace distance and 3) function IDs that track functions that have been covered.

3.5.1 Evalution Setup

In the experiments, we aim to answer the following questions:

RQ1 Is it worth applying static analysis?

RQ2 What are HAWKEYE's performance in reproducing crashes on target locations?

RQ3 How effective are the dynamic strategies in HAWKEYE?

RQ4 What are HAWKEYE's performance in reaching target locations?

Evaluation Dataset. We evaluated HAWKEYE with diverse real-world programs:

(1) **GNU Binutils** [66] is a set of binary analysis tools on Linux. This also serves a benchmark in other works such as [15, 16, 37].

(2) **MJS** [45] is a JavaScript interpreter for embedded devices. It is used to compare HAWKEYE directly with AFLGo due to implementation limitations of the latter.

(3) **Oniguruma** [53] is a versatile regular expression library used by multiple world famous projects such as PHP [67].

(4) **Fuzzer Test Suite** [68] is a set of benchmarks for fuzzing engines. It contains several representative real-world projects.

Evaluation Tools. We compare the following three fuzzers:

(1) **AFL** is the current state-of-the-art GBF. It discards all the target information for the target program and only utilizes the "basicblock transition" instrumentation.

(2) **AFLGo** is the state-of-the-art DGF based on AFL. Compared to AFL, it also instruments basicblock distance information.

(3) **HEGo** is the fuzzer whose basicblock level distance is generated with our static analysis procedure (Figure 3.3), but the dynamic fuzzing is conducted by AFLGo.

Here we mainly follow AFLGo's practice to only use AFL as the baseline for coverage oriented GBFs. Other techniques do not focus on directed fuzzing, and they are either orthogonal (e.g., CollAFL [18]) or may sometimes perform worse than AFL (e.g., AFLFast, as observed by [69]), or not easily comparable (e.g., libFuzzer [28]). The detailed reason is available on our website.

In the experiments, all AFL based fuzzers (AFL, AFLGo, and HEGo) are run in their "fidgety" mode [70]. For both AFLGo and HEGo, "time-to-exploitation" is set to 45 minutes for the fuzzer. Except for the experiments against GNU Binutils (Table 3.2) , where we follow exactly the setup in AFLGo [16] , all the other experiments are repeated 8 times, with a time budget of 4 hours. We use "time-to-exposure" (TTE) to measure the length of the fuzzing campaign until the first seed is generated that triggers a given error (in §3.5.3) or reaches a target location (in §3.5.4). We use *hitting round* to measure the number of runs in which a fuzzer triggers the error or reaches the target. For all the experiments, if the fuzzer cannot find the target crash within the time budget in one run, TTE is set to the time budget value.

Our experiments are conducted on an Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz with 28 cores, running a 64-bit Ubuntu 16.04 LTS system; during experiments, we use 24 cores and retain 4 cores for other processes.

3.5.2 Static Analysis Statistics

In Table 3.1, the first three columns denote the projects, programs and the sizes in their LLVM bitcode form. *ics* denotes the number of indirect call sites in the binary, which is calculated by counting those call sites without explicitly known callees; *cs* is the number of call sites; *ics/cs* denotes the percentage of indirect calls among all call sites. The next two columns denote the number where $C_N(f_1, f_2) > 1$ and $C_B(f_1, f_2) > 1$ between two adjacent functions f_1 and f_2, respectively. The last column denotes the time cost of call graph generation, which takes the majority of the time among all the directedness utility computation.

We can see from the table that the chosen benchmarks have fair diversities in terms of different metrics. It is also noticeable that the number of indirect function calls may contribute a large portion to the total number of function calls. Specifically, in libpng, 44.11% function calls are indirect function calls. This clearly shows the importance of building precise call graphs. Furthermore, the number of occurrences of $C_N(f_1, f_2) > 1$ and the number of occurrences of $C_B(f_1, f_2) > 1$ are also large, which shows the importance of taking into consideration the different patterns of call relations.

As to the overhead of the static directedness utility computation, except for *cxxfilt*, which requires approximately 12.5 minutes to generates the call graph, call graphs of most other projects can be generated in seconds. For cxxfilt, performance degradation lies in the inherent complexity of the project itself. From the program statistics in Table 3.1), it is obvious that the code base is bigger and the program structures are more complicated than the others. In fact, the bottleneck of the analysis is inside the pointer analysis implemented in SVF tool. We believe that it is worth the effort due to the fact that this procedure is done purely statically. And as long as the source code does not change, the call graph can be reused.

3.5.3 Crash Exposure Capability

The most common application of directed fuzzing is to try to expose the crash with some given suspicious locations that are supposed to be vulnerable, where the suspicious locations can be detected with the help of other static or dynamic vulnerability detection tools. In this experiment, we directly compare HAWKEYE with other fuzzers on some known crashes to evaluate its crash exposure capability.

3.5.3.1 Crash Reproduction against Binutils

In the beginning, we intended to compare GNU Binutils directly with AFLGo in our experiments since it is an important benchmark in [16] to demonstrate AFLGo's directedness. However, we found that the actual implementation of AFLGo has a few issues [71] in generating static distances. Most importantly, it takes too long to calculate the distances. As a result, when we tried AFLGo's static analysis on GNU Binutils 2.26, it failed to generate "distance.cfg.txt" which contains the distance information for instrumentation within 12 hours[1]. Although AFLGo can still perform fuzzing without distance information instrumentation, the fuzzing process is no longer *directed* without any distance input. Therefore, we reclaimed the results in [16] to compare with ours for the GNU Binutils benchmark [2]. We follow exactly the evaluation setup in [16] where each experiment is conducted for 20 times, with the time budget set as 8 hours; the initial input seed file only contains a line break (generated by `echo "" > in/file`). The target locations we specified are based on their CVE descriptions and the backtraces of the crashes. We compare HAWKEYE with AFLGo and AFL; the results are shown in Table 3.2. In AFLGo's paper, the A_{12} metric [72] is used to show the possibility that one fuzzer is better than the other according to all the runs. It is ignored in Table 3.2 since we cannot get the result of each run in their experiments.

[1] Besides the performance issue of AFLGo, another reason to reclaim AFLGo's results in Table 3.2 is that the hardware environments are similar. The reason is supported by the similar results produced by AFL in **[16]** and our experiments.

[2] Since CVE-2016-4487/CVE-2016-4488 and CVE-2016-4492/CVE-2016-4493 share the same target locations, we treat them as the same; the reclaimed value for CVE-2016-4487/CVE-2016-4488 are also average values.

TABLE 3.2: Crash reproduction in HAWKEYE, AFLGo, and AFL against Binutils.

CVE-ID	Tool	Runs	μ**TTE**(s)	Factor
2016-4487 2016-4488	Hawkeye	20	177	–
	AFLGo	20	390	2.20
	AFL	20	630	3.56
2016-4489	Hawkeye	20	206	–
	AFLGo	20	180	0.87
	AFL	20	420	2.04
2016-4490	Hawkeye	20	103	–
	AFLGo	20	93	0.90
	AFL	20	59	0.57
2016-4491	Hawkeye	9	18733	–
	AFLGo	5	23880	1.27
	AFL	7	20760	1.11
2016-4492 2016-4493	Hawkeye	20	477	–
	AFLGo	20	540	1.21
	AFL	20	960	2.01
2016-6131	Hawkeye	9	17314	–
	AFLGo	6	21180	1.22
	AFL	2	26340	1.52

We can observe the following facts: 1) For CVE-2016-4491 and CVE-2016-6131, HAWKEYE achieves the best results, with the most hitting rounds (both are 9 rounds) and the shortest μ**TTE** (18773s and 17314s). Compared with other tools, on average for both cases, HAWKEYE's improvements are significant in terms of hitting rounds ($> 20\%$) and μ**TTE** ($> 20\%$). 2) For CVE-2016-4487/4488 and CVE-2016-4492/4493, all tools reproduce the crashes in 20 runs, and HAWKEYE achieves the best μ**TTE**. Specifically, in these two cases, HAWKEYE's improvement in terms of μ**TTE** is significant — reducing at least 20% than other tools. 3) For CVE-2016-4489 and CVE-2016-4490, all tools reproduce the crashes in all runs within 7 minutes since these bugs are relatively easy to find. Apparently, in such cases, directed fuzzers have no significant advantage — in other words, when the crashes are shallow or easy to trigger, HAWKEYE's merits cannot show and fuzzing randomness matters for μ**TTE**.

To summarize, HAWKEYE has the real potential to fulfill directed fuzzing tasks where the target crashes are not easy to be detected.

3.5.3.2 Crash Reproduction against MJS

In order to *directly* compare the performance between HAWKEYE and AFLGo, we chose a project called *MJS*, which contains a single source file and the results are in Table 3.3. We used this project for direct comparison with AFLGo since AFLGo took too much time or failed to generate the distance information for other projects such as Oniguruma, libpng, etc. On *MJS*, AFLGo took an average of 13 minutes to generate the basicblock distance for different target locations. During experiments, the initial input seed files are all from the project's *tests* directory. The target locations are selected from the crashes reported in the project's GitHub pages, which correspond to four categories of vulnerabilities, namely *integer-overflow* (#1, https://github.com/cesanta/mjs/issues/57), *invalid-write* (#2, https://github.com/cesanta/mjs/issues/69), *heap-buffer-overflow* (#3, https://github.com/cesanta/mjs/issues/77), and *use-after-free* (#4, https://github.com/cesanta/mjs/issues/78). We can observe the following facts : 1) On #1 and #2, HAWKEYE achieves the best results, with the most hitting rounds and the shortest μ**TTE** for both cases. In terms of hitting rounds, HAWKEYE found #1's bug in 5 runs and #2's bug in 7 runs, while for the other two tools they only detected both crashes in 2 runs. Notably, this case is nontrivial and HAWKEYE reduces the μ**TTE** from about 3.5 hours to 0.5 hours. 2) On #3, for which all the tools reproduce the crash in 8 rounds. Still, HAWKEYE has a highly significant improvement on μ**TTE**, using less than one fourth μ**TTE** of other tools. 3) On #4, all the tools reproduce the crash in all rounds, and the μ**TTE** differences among them are not significant. As to A_{12}, we can see that HAWKEYE exhibits really good results, for example, the values in #2 are both 0.95, which means HAWKEYE has 95% confidence to perform better than both other tools.

3.5.3.3 Crash Reproduction against Oniguruma

Here we compare HAWKEYE with HEGo on Oniguruma to show the advantage of dynamic analysis strategies in HAWKEYE. Therefore, the major differences between HAWKEYE and HEGo is the dynamic part. Due to the aforementioned performance

TABLE 3.3: Crash reproduction in HAWKEYE, AFLGo, and AFL against MJS.

Bug ID	Tool	Runs	μTTE(s)	Factor	A_{12}
#1	Hawkeye	5	5469	–	–
	AFLGo	2	12581	2.30	0.77
	AFL	2	13084	2.39	0.77
#2	Hawkeye	7	1880	–	–
	AFLGo	2	12753	6.78	0.95
	AFL	2	12294	6.54	0.95
#3	Hawkeye	8	178	–	–
	AFLGo	8	819	4.60	0.91
	AFL	8	1269	7.13	0.95
#4	Hawkeye	8	5519	–	–
	AFLGo	8	5878	1.07	0.57
	AFL	8	5036	0.91	0.48

issues of AFLGo in static analysis on Oniguruma and other big projects, hereinafter, we use HEGo as an alternative to AFLGo in the subsequent experiments. We therefore compare HAWKEYE with HEGo to show the effectiveness of our dynamic strategies.

In Table 3.4, we compare HAWKEYE with HEGo and AFL against the Oniguruma regex library. The first three bugs come from the reported CVEs which occur on version 6.2.0, and Bug #4 is a newly fixed vulnerability issue on its GitHub pages. Some observations are: 1) For #3 and #4, HAWKEYE achieves the best results among the three tools. Especially, for #4, the improvements in both hitting rounds and μ**TTE** are highly significant. For #3, HAWKEYE can find the bug in one more round, but the μ**TTE** is similar for the three tools. 2) For #1 and #2, all the tools reproduce the crash in 8 rounds. On the other hand, HAWKEYE and HEGo have no significant differences in μ**TTE**. From the results, we can conclude that the dynamic analysis strategies used in HAWKEYE are effective.

3.5.4 Target Location Covering

Certain locations in the target programs are hard to reach, though they may not trigger crashes. In practice, the generated seeds that can cover such locations can also be used as initial seeds for GBFs to boost their coverage. Therefore, this criterion is an important factor for measuring DGFs' capabilities.

TABLE 3.4: Crash reproduction in HAWKEYE, HEGo, and AFL against Oniguruma.

Bug ID	Tool	Runs	μTTE(s)	Factor	A_{12}
#1	Hawkeye	8	139	–	–
	HEGo	8	149	1.07	0.58
	AFL	8	135	0.97	0.54
#2	Hawkeye	8	186	–	–
	HEGo	8	228	1.23	0.88
	AFL	8	372	2.00	1.0
#3	Hawkeye	2	13768	–	–
	HEGo	1	14163	1.03	0.56
	AFL	1	14341	1.04	0.57
#4	Hawkeye	7	6969	–	–
	HEGo	3	12547	1.80	0.82
	AFL	1	14375	2.06	0.88

Google's fuzzing test suite contains three projects that specially focus on testing fuzzers' abilities to discover hard-to-reach locations, namely *libjpeg-turbo-07-2017* (#1), *libpng 1.2.56* (#2, #3) and *freetype2-2017* (#4). In these benchmarks, the target locations are specified by file names with line numbers in the source files. We manually added additional "sentinel" code in the target locations ("exit(n)", where the values of "n" distinguish these locations) to indicate that the specified locations are reached.

Table 3.5 shows the results on these benchmarks. Case #1 and #4 show that HAWKEYE exhibits good capability in terms of rapidly covering the target locations, according to μ**TTE** and the factor columns; considering A_{12}, the behaviors are also steady. In case #2 and #3, it takes little time to reach these target locations for all the fuzzers. While on the relevant project page [73], it is mentioned clearly that they "currently require too much time to find". We actually tried this benchmark on libFuzzer with the default scripts in 2 machines, indeed it failed to reach the target locations. This root cause of the inconsistency may lie in the fact that the inner mechanisms may affect the actual fuzzing effectiveness (In fact, libFuzzer is known to be quite different from AFL and its derivations). The other observation is that HEGo has a rather big value in terms of μ**TTE** compared to other tools. It turns out that in one of the runs, the TTE is 524s, much larger than all the other runs.

It is worth noting that the μ**TTE** to cover the target locations (Table 3.5) is *quite* different from the μ**TTE** to trigger real-world crashes (Table 3.4): the TTEs of the former are calculated based on the duration to cover the specific *line* at the first time; while the

TABLE 3.5: Target location covering results in HAWKEYE, HEGo and AFL against libjpeg-turbo, libpng, freetype2 (Fuzzer Test Suite).

ID	Project	Tool	Runs	μ**TTE**(s)	Factor	A_{12}
#1	jdmarker.c:659	HAWKEYE	8	1955	–	–
		HEGo	8	2012	1.03	0.53
		AFL	8	4839	2.48	0.95
#2	pngread.c:738	HAWKEYE	8	23	–	–
		HEGo	8	16	0.70	0.43
		AFL	8	130	5.65	1.00
#3	pngrutil.c:3182	HAWKEYE	8	1	–	–
		HEGo	8	66	66.00	0.56
		AFL	8	3	3.00	0.51
#4	ttgload.c:1710	HAWKEYE	7	4283	–	–
		HEGo	7	4443	1.04	0.55
		AFL	6	5980	1.40	0.60

TTEs of the latter are tightly relevant to *branch coverage* or even *path coverage* since typically bugs in widely used software can only be triggered with special path conditions and it requires covering a few execution traces. Although Table 3.5 shows that HAWKEYE's improvements against HEGo in covering target locations are not obvious (and for a few cases, it performs worse), we can observe in Table 3.4 the acceleration on crash reproduction is significant especially for #4 (1.80x, nearly 2 hours off) and #2 (1.23x). This indicates that dynamic strategies are quite effective in *detecting crashes*.

3.5.5 Answers to Research Questions

With the experiments conducted in Tables 3.2, 3.3, 3.4 and 3.5, we can answer the research questions.

RQ1 We consider it is worth to apply static analysis. As shown in Table 3.1, the time cost of our static analysis is generally acceptable compared to the runtime cost during fuzzing. Even for the cxxfilt cases in Table 3.2, which takes on average 735 seconds, HAWKEYE outperforms the vanilla AFL in most of the cases. Two notable results are CVE-2016-4491 and CVE-2016-6131, it saves roughly 2000s and 9000s to detect the crash; as shown from the A_{12} metric, the results are also consistent in all the 20 runs. On the other hand, HAWKEYE also demonstrates some boosts for fuzzing.

RQ2 HAWKEYE performs quite well in detecting crashes. From the results in Tables 3.2, 3.3 and 3.4, we can clearly see that HAWKEYE can detect the crashes more quickly than all the other tools; the results are even steady among different runs as shown by different A_{12} results.

RQ3 The dynamic strategies used in HAWKEYE are quite effective. It is obvious that in all the experiments we conducted, HAWKEYE outperforms the others. In particular, the experiments in comparison with HEGo (Table 3.4 and 3.5) show that our combination of power scheduling, adaptive mutation strategies, and seed prioritization make HAWKEYE converge faster than AFLGo's simulated annealing based scheduling.

RQ4 From the results in Table 3.5, we are confident that HAWKEYE has the capability to reach the target locations rapidly.

In practice, HAWKEYE also demonstrates its power in exposing crashes with the help of other vulnerability detection tools. For example, for Oniguruma and MJS projects, with the Clang Static Analyzer [74] (the builtin and our customized checkers) reporting suspicious vulnerability locations (i.e., target locations) in the programs, HAWKEYE successfully detected the crashes by directing the fuzzing to those locations. Interestingly, for MJS, we marked several of the authors' newly patched program locations, and detected a few other crashes even further. As a result, HAWKEYE has reported more than 28 crashes in projects Oniguruma and MJS. We have also found multiple vulnerabilities in other projects such as Intel XED x86 encoder decoder (4 crashes), Espruino JavaScript interpreter (9 crashes). All these crashes have been confirmed and fixed, and 15 of them have been assigned with CVE IDs.

3.5.6 Threats to Validity

The internal threats of validity are twofold: 1). Several components of HAWKEYE (e.g., Algorithm 2 and 3) utilize the predefined thresholds to make decisions. Currently, these thresholds (e.g., $\gamma = 0.1$, $\delta = 0.4$, $\sigma = 0.2$, $\zeta = 0.8$) are configured according to our preliminary experiments and previous experience in fuzzing. Systematic research

will be planned to investigate the impact of these thresholds and figure out the best configurations. 2). As we rely on the lightweight program analysis tools like LLVM and SVF [65] to calculate the distance, issues of these tools may affect the final results. To overcome this, enhancing HAWKEYE with other tools will be an alternative solution.

The external threats rise from the choice of evaluation dataset and the CVEs for crash reproduction. Despite we adopt the program Binutils that is used in AFLGo [16], the evaluation results still need to be generalized with an empirical study on more projects in the future. Besides, the tested CVEs in *MJS* and *Oniguruma* are *not selectively* chosen for the purpose to show the advance of HAWKEYE— we pick them since they are reported within a recent period.

3.6 Related Work

The work in this chapter is related to the following lines of research:

3.6.1 Directed Greybox Fuzzing

Some other DGF techniques have been proposed besides HAWKEYE. AFLGo [16] is the state-of-the-art directed greybox fuzzer which utilizes a simulated annealing-based power schedule that gradually assigns more energy to seeds that hold the trace closer to the target locations. In AFLGo, the authors proposed a novel idea of calculating the distance between the input traces and the target locations. This is a good starting point by combining such target distance calculation with greybox fuzzer. HAWKEYE is inspired by AFLGo however provides significant improvements on both the static analysis and dynamic fuzzing. As shown in §3.5, HAWKEYE generally outperforms AFLGo in terms of reaching the target locations and reproducing crashes, thanks to embedding in-depth consideration about the four desired properties into the design. SeededFuzz [75] uses various program analysis techniques to facilitate the generation and selection of initial seeds which helps to achieve the goal of directed fuzzing. Equipped with the improved seed selection and generation techniques, SeededFuzz can reach more critical

locations and find more vulnerabilities. The core techniques of SeededFuzz are orthogonal to HAWKEYE because SeededFuzz focuses on the quality of initial seed inputs while HAWKEYE focuses on the four desirable properties regardless of initial seeds.

Note that our proposed four properties can also be applied for DGF on programs where the source code is unavailable. In fact, we are extending HAWKEYE to be able to work on the binary-only fuzzing scenarios. Technically, the target locations can be determined by binary-code matching techniques on attack surface identification [76,77]; the static analysis can be achieved with binary analysis tools such as IDA [78]; and the instrumentation can be done by dynamic binary instrumentors such as Intel Pin [36]. We envision the extended HAWKEYE can piggyback on these techniques and demonstrate its effectiveness even further.

3.6.2 Directed Symbolic Execution

. Directed Symbolic Execution (DSE) is one of the most related techniques to DGF as it also aims to execute target locations of the target program. Several works have been proposed for DSE [31, 79–82]. These DSE techniques rely on heavyweight program analysis and constraint-solving to reach the target locations systematically. A typical example of DSE is Katch [82], which relies on symbolic execution, augmented by several synergistic heuristics based on static and dynamic program analysis. Katch can effectively find bugs in incomplete patches and increase the patch coverage comparing to the manual test suite. However, as discussed in [16], DGF is generally more effective in real-world programs as DSE techniques suffer from the infamous path-explosion problem [83]. In contrast to DSE, HAWKEYE relies on lightweight program analysis, which ensures its scalability and execution efficiency.

Taint Analysis Aided Fuzzing. Taint analysis is also widely used to facilitate directed white-box testing [19, 37, 84–86]. The key intuition of using taint analysis in fuzzing is to identify *certain parts* of the input which should be mutated with priority. In such a way, the fuzzer can drastically reduce the search space for reaching certain desired locations. Taint based approaches are more scalable than the DSE techniques and can help the fuzzer to reach certain preferable locations such as rare branches in Fairfuzz [37] or

checksum related code in TaintScope [86]. Different from HAWKEYE, these techniques are not fed with *given target locations* (e.g., file name and line numbers) but based on source-sink pairs. Thus, such techniques do not have advantages in scenarios where the target locations are absolutely clear, such as patch testing and crash reproduction.

3.6.3 Coverage-based Greybox Fuzzing

. The purposes of coverage-based greybox fuzzing (CGBF) and DGF are different. However, some techniques proposed to boost the performance of CGBF could also be adopted by HAWKEYE. For example, CollAFL [18] utilizes a novel hash algorithm to solve AFL's instrumentation collision problem. Skyfire [21] learns a probabilistic context sensitive grammar (PGSG) to specify both syntax features and semantic rules, and then the second step leverages the learned PCSG to generate new seeds. Xu *et al.* [87] proposed a set of new operating primitives to improve the performance of greybox fuzzers. Another important topic in CGBF is about guiding the fuzzer through path constraints. [19, 22, 83, 85, 88] aim to help the CGBFs to break through path constraints. Moreover, Orthrus [69] applies static analysis on AST, CFG, and CG to extract complicated tokens via customizable queries. HAWKEYE can benefit through combining with the aforementioned techniques.

3.7 Conclusion

We propose a novel directed greybox fuzzer, HAWKEYE. The design of HAWKEYE embeds four desired properties for directed fuzzing by combining static analysis and dynamic fuzzing in an effective way. Equipped with a better evaluation of the distance between input execution traces and the user specified target locations, HAWKEYE can precisely and adaptively adjust its seed prioritization, power scheduling as well as mutation strategies to reach the target locations rapidly. A thorough evaluation showed that HAWKEYE can reach the target locations and reproduce the crashes much faster than existing state-of-the-art greybox fuzzers. The promising results indicate that HAWKEYE can be effective in patch testing, crash exposure, and other scenarios.

Chapter 4

DOUBLADE: Thread-aware Greybox Fuzzing

4.1 Introduction and Motivation

Multithreading as a popular programming paradigm is an effective way of utilizing multi-core computation resources in modern processors. Meanwhile, the *non-deterministic* behaviors caused by thread-interleavings also pose substantial challenges to bug detection in multithreaded programs [89]. On one hand, the interleavings across threads inherently introduce multiple subtle concurrency bugs, e.g., data races, atomic violations, and deadlocks. These bugs can cause undefined program behaviors and sometimes can induce vulnerabilities such as use-after-free (CVE-2016-1972, CVE-2018-5873) and heap-buffer-overflow (CVE-2017-8244), denial-of-service (CVE-2018-0381). On the other hand, as multithreaded programs may accept some inputs, some bugs or vulnerabilities may be difficult or impossible to be triggered with limited initial seed inputs, regardless of whether such bugs are caused by concurrency bugs. Compared to the lower level vulnerabilities that root from buffer-overflow, invalid-memory-read, etc, multithreading-relevant bugs can result from logic errors that may or may not cause crashes.

Categories of Multithreading-relevant Vulnerabilities and Bugs Figure 4.1 depicts the multithreading relevant vulnerabilities and bugs. The rectangle denotes all

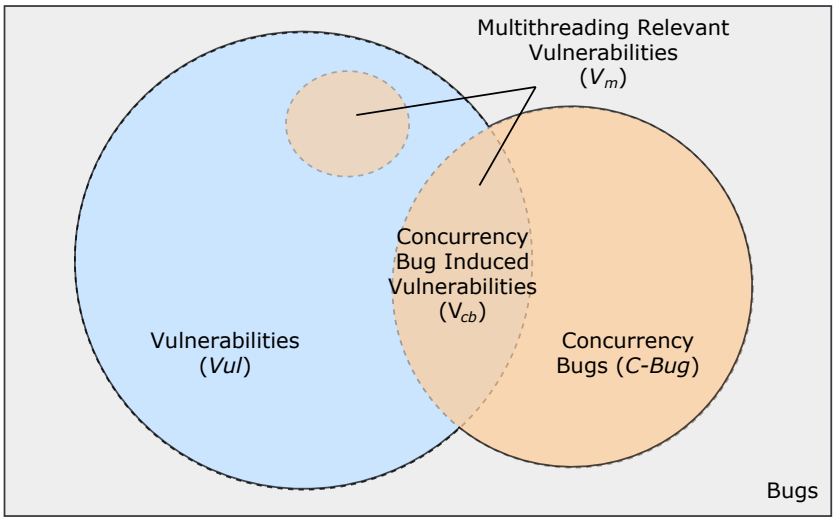

FIGURE 4.1: Venn Diagram of the Multithreading Relevant Vulnerabilities and Bugs.

categories of bugs. The left big circle denotes all categories of vulnerabilities (*Vul*); the right circle denotes concurrency-bugs (C_{bug}); the intersection of *Vul* and C_{bug} is termed as *concurrency-bug induced vulnerabilities* under the notation V_{cb}. V_m denotes *multithreading-relevant vulnerabilities* that may occur in executing multithreading code segments of the program; it is the union of V_{cb} and the smaller circle inside *Vul*. Consequently, V_s=*Vul*-V_m denotes the vulnerabilities that do not occur in multithreading executions. It is worth noting not all V_m vulnerabilities are *concurrency-bug induced vulnerabilities* (V_{cb}). We group vulnerabilities like CVE-2015-6761 and CVE-2016-8438 into V_m but not into V_{cb} since their root causes are not from thread-interleavings. V_m also contains vulnerabilities that occur in multithreading execution but actually because of faults, e.g., *out-of-bound-read* and *integer-overflow*, in the individual thread. Since the root causes of real-world multithreading vulnerabilities are notoriously complicated [89], we do not differentiate whether they belong to V_{cb} or V_m-V_{cb}, but generalize them with V_m based on their occurrences. We will show later this treatment is more natural for fuzzing purposes.

In Listing 4.1, in the multithreading execution environment, if the condition *satisfy(s_var)* returns true, it executes *func_1*; otherwise *func_2*. Let us suppose that there is a vulnerability inside *func_2* that is not introduced by concurrency bugs. Note that this branching results may rely on the number of threads, the input content themselves.

```
1  int compute(void *s_var) {
2      ...
3      if (satisfy(s_var)) {
4          func_1(s_var);
5      } else {
6          func_2(s_var);
7      }
8      ...
9  }
10
11 int main(int argc, char** argv) {
12     u8* input = read_file(argv[1]);
13     validity_check_1(input);
14     pthread_t t1, t2;
15     pthread_create(t1, compute, input[0]);
16     pthread_create(t2, compute, input[128]);
17 }
```

LISTING 4.1: An Example illustrating the vulnerability lies in V_m but not V_{cb}.

For example, if in single-threaded mode, *satisfy(s_var)* always returns true, the vulnerability will never be triggered; this vulnerability can only be revealed with more than one thread. Under other circumstances, suppose for the input content, *func_2* will not be executed except for some very specific conditions (e.g., error handling), the vulnerabilities can only be revealed with certain content. Therefore, it may not be enough to only consider the C-Bug induced vulnerabilities V_{cb}.

As mentioned in §2.1, a conventional GBF usually has no awareness of multithreading. In fact, the only symptom it observes is that a seed has non-deterministic behaviors in the sense that the same seed exercises different traces during calibration execution. However, since *multiple reasons* can cause non-deterministic behaviors, a GBF typically does not have any countermeasures: all it can do is to increase the calibration value $\mathbb{N}_c$ (see Algo. 1) to execute the seeds more times to get more stable running statistics.

When fuzzing a multithreaded program, as long as the seeds can execute the multithreading segments, it always exhibits certain non-deterministic behaviors due to the interleavings across different threads. Compared with the seeds that cannot even enter the multithreading segments, we would *prioritize* these seeds that have thread-interleavings since it is these seeds that 1) may themselves introduce vulnerabilities with different interleavings 2) are more likely to be mutated to seeds that can exercise similar paths [20].

On the other hand, preserving more multithreading relevant seeds will be more helpful for concurrency bug detectors to detect concurrency violations. Therefore, to enhance the effectiveness of greybox fuzzing on multi-threading programs, we should provide *thread-aware* analyses to help exercise more thread-interleaving paths and generating more multithreading relevant seeds.

We hereby present DOUBLADE, a new greybox fuzzing technique for multithreaded programs. The core of DOUBLADE is a novel thread-aware seed generation technique which effectively produces valuable seeds to test the multithreading context. DOUBLADE relies on a set of thread-aware instrumentation methods consisting of a stratified exploration-oriented instrumentation and two complementary instrumentation. The dynamic strategies are hereby optimized for the feedback provided by these instrumentations to improve the effectiveness of fuzzing.

The experimental results demonstrate DOUBLADE significantly outperforms the state-of-the-art greybox fuzzer AFL [27] in generating multithreading relevant seeds, detecting multithreading relevant vulnerabilities, and exposing concurrency bugs via generated seeds. In particular, DOUBLADE detected 9 multithreading relevant vulnerabilities and 2 of them have been assigned CVE IDs. Additionally, DOUBLADE helped to expose 19 new concurrency bugs with the generated seeds.

The contributions of this chapter are as follows:

1) We designed a novel stratified selective instrumentation strategy specifically designed for exploring thread-interleaving induced paths. This instrumentation significantly increases both the total number of multithreading relevant seeds and its percentage among all generated seeds.

2) We introduced two other thread-aware instrumentations to exhibit more thread-interleavings as well as distinguish overall threading context. This helps increase the thread-interleaving induced coverage during calibration execution, as well as diversify the generated seeds.

```
1  int g_var=1;
2  void inc(int *pv) { *pv += 2; }
3
4  void check(char * msg) {
5    if (msg[0] <= '1') exit(1);
6  }
7
8  int compute(void *s_var) {
9    *s_var += 3;
10   *s_var *= 2;
11   if (*s_var<2) inc(&g_var);
12   pthread_mutex_lock(&m);
13   inc((int*)(s_var));
14   pthread_mutex_unlock(&m);
15   return *s_var;
16 }
17
18 int main(int argc, char **argv) {
19   check(argv[1]);
20   pthread_t T1, T2;
21   pthread_create(T1,NULL,compute,argv[1]);
22   pthread_create(T2,NULL,compute,argv[1]);
23   ...
24 }
```

LISTING 4.2: A program abstracted from real-world multithreaded programs.

3) We integrated the dynamic fuzzing strategies with these thread-aware instrumentations and implemented our enhanced GBF, DOUBLADE, for fuzzing multithreaded programs. To the best of our knowledge, this is the first greybox fuzzer that is optimized for detecting vulnerabilities and concurrency bugs in multithreaded programs.

4.2 Issues in Fuzzing Multithreaded Programs

4.2.1 A Running Example

Listing 4.2 is a program abstracted from real-world multithreaded programs such as *GraphicsMagick*. Before processing the input, it does a validation check inside *check* against some properties of *argv*[1]. If the check fails, the program terminates immediately. The core functionality starts from two threads created at lines 21 and 22 respectively in *main* function. There are two shared variables: 1) the global variable

`g_var` has an initial value 1, 2) and input `argv`[1] is passed to both threads T1 and T2 through the thread-forking call to *pthread_create*. Apparently, there are multiple reads and writes on `g_var` and `argv`[1] and this program suffers from data races.

With this example, we would like to demonstrate that the state-of-the-art GBFs such as AFL can be ineffective in exploring thread-interleaving relevant paths due to *unawareness of multithreading*.

4.2.2 No Strategies to Track Thread-Interleavings

GBFs such as AFL typically instrument new instructions into the target programs to collect runtime coverage information. Specifically, AFL treats the entry instruction of each basicblock as the basicblock's *deputy*. From now on, we will refer AFL's default selection strategy over the *deputy* instruction as AFL-Ins. During calibration execution, AFL labels a calculated value to each transition that connects the *deputies* of two consecutively executed basicblocks [64]. By maintaining a set of values for queued seeds, AFL *tracks* the "coverage" of a target program. Procedure *cov_new_trace* in Algorithm 1 is to check whether a value has already been contained in the set.

Figure 4.2b depicts the transitions upon executing the functions *compute* and *inc*. For brevity, we use source code to explain the problem and use *statements* to represent the lower level *instructions* in assembly or LLVM IR [33]. The arrows denote the transitions between *all* the statements. The pentagons denote the first statements of basicblocks while the other statements are represented by rectangles. Since AFL-Ins only cares about the first statements, only the branching edges from `if(s_var<2)` and the two function call edges to *inc* are bookkept – these transitions are marked as solid arrows.

AFL-Ins works well on single-threaded programs: the kept transitions can reflect both branching conditions (e.g., "`if(s_var<2)` → `inc(&g_var)`") and the function calls (e.g., "`inc((int*)(s_var))` →`*pv+=2`"). However, in the presence of multithreading, it may frequently miss several important transitions of the interleavings between concurrently executed threads. Let us focus on two *different* seeds' execution at

two statements: ①:"`*s_var+=3`" and ②: "`*s_var*=2`". Threads T1 and T2 may execute the two statements with these interleavings: the two T1:①→T2:①→T1:②→T2:②, or T1:①→T1:②→T2:①→T2:②. After the second ② is executed the value of `*s_var` is likely to be different; in addition, dependent on the initial value of `*s_var` (propagated from `argv[0]`), the condition `s_var<2` may also be affected. Therefore, these interleavings may affect subsequent executions and it is expected to keep track of them. However, since AFL only tracks the *deputy-to-deputy* transitions, it merely observes that there is a transition from ①→①. Since AFL keeps seeds only when it "sees" a new transition, it simply discards the second one. However, since the second seed indeed executes a different interleaving, keeping it usually brings positive coverage feedback [20, 90]. In this sense, it is preferable to apply certain instrumentation to explore more thread-interleaving paths and keep more multithreading relevant seeds.

4.2.3 Unawareness of Threading Context

AFL knows nothing about the threading context, therefore it cannot distinguish whether a transition is from T1 to T2 or from T1 to T1. For example, as to the (four) interleavings that can occur when executing ① and ②, AFL is only aware of ①→①. Since such a transition can also happen when ① is inside a loop, AFL may think this is not a "new" transition. Furthermore, the threading information does help to provide additional feedback that is orthogonal to the transitions between the deputy instructions. In this sense, we should provide a strategy to record the threading context.

4.2.4 Low Diversity across Executions

When encountering a non-deterministic behavior seed, the only strategy of AFL is to execute the seeds more times. However, since the calibration execution is applied *continuously* $\mathbb{N}_c$ times, the systematic level environment such as CPU usage, memory consumption, or other I/O status is prone to be similar [91, 92]. This will decrease the entropy to diversify the actual scheduling. For example, in a calibration execution with $\mathbb{N}_c = 40$, T1:①→T2:①→T1:②→T2:② and T2:①→T1:①→T2:②→T1:② may occur 15

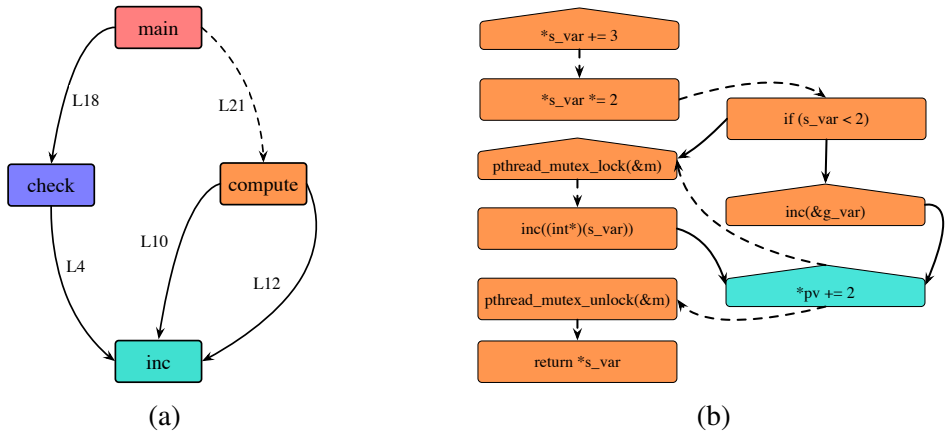

FIGURE 4.2: Thread-aware Callgraph (4.2a) of Listing 4.2 and its edge transitions in functions `compute` and `inc` (4.2b).

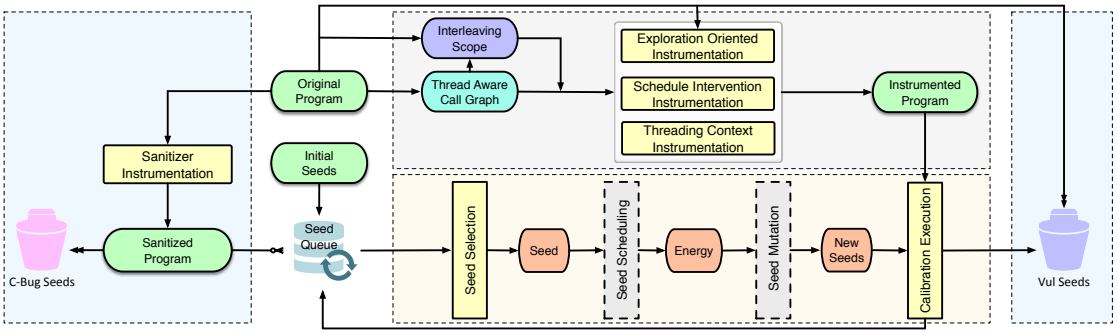

FIGURE 4.3: Overall workflow of DOUBLADE.

and 20 times respectively, however interleavings such as T1:①→T1:②→T2:①→T2:② only occurs 5 times, while there is no T2:①→T2:②→T1:①→T1:② at all. In addition, since the running statistics are collected during calibration execution, this would also affect the decision in seed triaging. Ideally, we would like to keep as many distinct interleavings as possible since that marks the potential interleavings a seed can exhibit with different scheduling priorities.

4.2.5 Our Solutions

In order to improve the effectiveness of fuzzing multithreaded programs, we propose the following solutions.

S1 Instead of equally choosing the entry instruction of a basicblock as the deputy instruction, we apply a stratified exploration-oriented instrumentation by distinguishing whether an instruction may happen-in-parallel with others [93, 94]. This

helps the fuzzer to track more thread-interleaving transitions.

S2 We instrument at the call site of thread-forking, locking, unlock, joining APIs and collect the thread and deputy instruction information to calculate the overall threading context per execution. This information is utilized to diversify the threading context of the seeds in the queue.

S3 We instrument at the entry of a thread start routine function a segment that can dynamically adjust the thread priority. It aims to increase the interleaving diversity during calibration execution.

4.3 System Overview

Figure 4.3 depicts the overall workflow of DOUBLADE. It contains three major components: (1) static-analysis-guided instrumentation (shown in the *top center* area), (2) dynamic fuzzing (shown in the *bottom center* area), and (3) concurrency bug replay (shown in the *left and right* area). During instrumentation (§ 4.4), we first compute the thread-aware ICFG (interprocedural control-flow graph) in § 4.4.1.1 for a multithreaded program ($\mathbb{P}_o$). Based on this ICFG, we perform three categories of instrumentation: the one focusing on exploring more multithreading relevant paths (§ 4.4.2), the one aiming to intervene the thread scheduling (§ 4.4.3) and the one that provides the threading context (§ 4.4.4) to eventually produce the instrumented program $\mathbb{P}_f$. During dynamic fuzzing § 4.5, based on the instrumentation feedback, we apply seed selection (§ 4.5.1), calibration execution (§ 4.5.2) and seed triaging (§ 4.5.3) that are turned for generating multithreading relevant seeds. Since the fuzzing procedure can only detect *vulnerabilities* that are resulted from crashed seeds, we need a concurrency bug detector to replay with the generated seeds to see whether they can trigger certain concurrency violations; this will be covered in the evaluation section (§ 4.6.2).

4.4 Static Instrumentation

The instrumentation is used to provide the thread-aware information for fuzzing.

As discussed in §4.2.4, the instrumentation strategy in existing GBFs is thread-unaware, which may miss transitions caused by thread-interleaving for triggering vulnerabilities in multithreaded programs. Drawn from this insight, the precise fuzzing should distinguish program statements that may be accessed by concurrent threads, which requires "finer-grained" instrumentation to pay attention to the seeds that can capture the instruction transitions caused by interleaved threads, while deprioritize the seeds that exercise new paths only within a single thread. We use results from static context-sensitive may-happen-in-parallel analysis [94] to determine the thread-interleaving scope. The pointer analysis and lock analysis are enabled to further refine the scope in order to filter out the interleaving-free statements. The static analysis is adopted as the preprocessing in §4.4.1 for three instrumentations (§ 4.4.2-§ 4.4.4) to explore new paths, intervene the schedule, and guide the seed selection procedure.

4.4.1 Preprocessing

4.4.1.1 Thread-aware ICFG Generation

We firstly apply Andersen's inclusion-based pointer analysis [57] on the target program. The points-to results are used to resolve indirect calls to eventually produce an interprocedural control-flow graph (ICFG). By taking into account the semantics of pthread APIs, we get an ICFG that is aware of the following multithreading information:

(1) `TFork` is the set of program sites that call thread-forking functions. This includes the explicit call to *pthread_create* or the *std::thread* constructor that internally uses *pthread_create*, etc. These called functions, denoted as F_{fork}, are extracted from the semantics of these forking sites.

(2) `TJoin` contains call sites for functions that mark the end of a multithreading execution. It includes the call sites of the pthread APIs such as *pthread_join*, *pthread_cancel*, *pthread_exit*, etc.

(3) `TLock` is the set of sites that call thread-locking functions such as *pthread_mutex_lock*, *pthread_mutex_trylock*, etc.

(4) `TUnLock` is the set of sites calling thread-unlocking functions such as *pthread_mutex_unlock* and *pthread_rwlock_unlock*.

(5) `TShareVar` is the set of variables that *may be* shared among different threads. This includes the global variables and those variables that are passed from the threading fork sites `TFork`.

4.4.1.2 Suspicious Interleaving Scope Extraction

Given a program that may run with multiple threads, we would prefer the instrumentation to collect various traces to reflect the interleavings. However, instrumentation always brings execution overhead to original programs, especially when adopting an intensive instrumentation strategy to cover all concerned sites. Owing to the static information provided by the thread-aware ICFG in §4.4.1, we know that multithreading interleavings may only happen on *certain* program statements. We hereby use L_m to denote the set of these statements and name it as *suspicious interleaving scope*. Thus, the instrumentation is targeted on these statements. A statement l is added into an thread-interleaving scope if there exists another statement l', which may happen-in-parallel with l and may access the same memory inferred by the pointer analysis, but not protected by the same lock.

In Listing 4.2, $F_{fork} = \{compute\}$ based on the function call at Lines 21 and 22. We then get all the functions that may be (directly or indirectly) called by functions inside F_{fork}, i.e., $\{inc, compute\}$ and the scope L_m comes from Lines 1, 2, 9∼15. We exclude the statements that do not access or modify the shared variables g_var1, g_var2, s_var, which means we should exclude Lines 12 and 14. In the end, the scope is determined as $L_m = \{1, 2, 9, 10, 11, 13, 15\}$. It is worth noting that line 13 *is* inside L_m since it may happen-in-parallel with lines 9 and 10.

4.4.2 Exploration-Oriented Instrumentation

With the knowledge of L_m, we are able to instrument more stubs on these instructions inside the scope than the others, for exploring new "transitions". However, it is *impractical* to instrument intensively on *each instruction* inside L_m since this still may greatly reduce the overall performance. It is also *unnecessary* to do so — although theoretically interleavings may happen everywhere at each instruction inside L_m, in practice several instructions in one thread can still be executed subsequently. This means that we can skip some instructions for instrumentation, or instrument the instructions *with a probability*. We still need to instrument on the other parts of the program in case that the initial seeds do not cover the multithreading statements at all. Hence, we need the instrumentation to help explore more *new* traces. Also similarly, we can *skip* instrumentation on some instructions with certain probabilities.

4.4.2.1 Instrumentation Probability Calculation

We first calculate a *base instrumentation probability* according to cyclomatic complexity. We use this since it has been demonstrated that bugs or vulnerabilities usually come from functions that have higher cyclomatic complexity [95–97]. For each function f, we firstly calculate the complexity value: $M_c(f) = E(f) - N(f) + 2$ where $E(f)$ is the basicblock edge number of the function, $N(f)$ is the number of basicblocks. Intuitively, this value indicates the complexity of the function across its basicblocks. As 10 is considered to the preferred upper bound [95], we determine the base probability as

$$P_e(f) = \min\left\{\frac{E(f) - N(f) + 2}{10},\ 1.0\right\} \tag{4.1}$$

We use P_s as the probability to *selectively* instrument on the first instruction of a basicblock that is *totally* outside the scope of multithreading environment, i.e., *none of* the instructions inside this basicblock belongs to L_m.

$$P_s(f) = \min\left\{P_e(f),\ P_{s0}\right\} \tag{4.2}$$

where $0 < P_{s0} < 1$. P_s0 should not be set too large since that would bring instumentation overhead; it should not be set too small since otherwise $P_s(f)$ would always equal

P_{s0}, failing to consider the complexity influence of the underlying function. Empirically, we set $P_{s0} = 0.5$.

On the other hand, for each basicblock b inside the given function f, we calculate the total number of instructions $N(b)$, and the total number of memory operation instructions $N_m(b)$, e.g., load/store operations, or builtin memory function calls such as *memcpy*, *strncat*, *malloc*, *free*, etc. Then for each of the instructions, we instrument based on the probability as:

$$P_m(f,b) = \min\left\{P_e(f) \cdot \frac{N_m(b)}{N(b)},\ P_{m0}\right\} \tag{4.3}$$

where P_{m0} is a factor satisfying $0 < P_{m0} < 1$ and defaults to 0.33. Note the default value of P_{m0} follows AFL's practice to avoid heavy instrumentations.

4.4.2.2 Instrumentation Algorithm

The exploration-oriented instrumentation algorithm is described in Algorithm 5. It iterates each function inside a program. For each basicblock b in function f, we firstly get the intersection of the instructions inside b and L_m. If this intersection L_m (b) is empty, it instruments the *first instruction* of b with a probability of P_s (f). Otherwise, for the first instruction in b, we always instrument it (instrument with a probability of 1.0); for the other instructions, if they are inside L_m, we instrument them with a probability of $P_m(f,b)$. We will refer our selection of deputy instructions as MT-Ins.

4.4.2.3 Take-aways for MT-Ins

Firstly, for both AFL-Ins and MT-Ins, the coverage is shaped by the transitions between deputy instructions. The difference is that we use a stratified instrumentation strategy to focus on the interleavings introduced by multithreading. Compared to AFL-Ins, MT-Ins instruments less on single threading program statements, while applies more instrumentations to explore interleavings across different threads with certain probabilities.

Secondly, mistakenly instrumenting on non-entry instructions that are not in L_m will *only introduce runtime overhead*, but *never* help to keep more seeds. Due to this, we

Algorithm 5: Exploration-Oriented Instrumentation

input : Target program P, suspicious interleaving scope L_m
output: Program P instrumented with exploration stubs

```
1  for f ∈ P do
2      for b ∈ f do
3          L_m(b) = L_m ∩ b;
4          if L_m(b) != ∅ then
5              for i ∈ b do
6                  if is_first_instr(i, b) then
7                      exploration_instrument(i, 1.0);
8                  else if i ∈ L_m then
9                      exploration_instrument(i, P_m(f, b));
10         else
11             for b ∈ f do
12                 i = get_first_inst(b);
13                 exploration_instrument(i, P_s(f));
```

may expect that there will not be seeds *accidentally* added into the seed queue due to over-estimation of L_m.

Thirdly, MT-Ins's strategy to tracking multithreading interleaving *is* useful and it is worth instrumenting more on these instructions that may be executed parallelly with other instructions. The major reason is that we can expect to have more valuable seeds. Same as other GBFs, we still rely on observations of "new transitions" to keep seeds, and MT-Ins tends to keep more seeds that execute thread-interleaving, regardless of the precision of the calculated L_m. First, it means that more seeds in the seed queue for fuzzing have executed the multithreading code segments, instead of having failed the validation check quite earlier. Hence, these seeds can execute the "deeper part" of a multithreaded program. When they are mutated, compared with those seeds that fail the validity check, there is a higher chance that the newly generated seeds from these seeds also pass the check and exercise different parts of the multithreading segments. Therefore, the high ratio of seeds that can execute multithreading segments already indicates the seeds are probably of high quality. Second, since seeds usually exhibit some preference in executing with certain interleavings, some thread-interleavings are more "stable", they may be just like the regular transitions inside a single thread.

Lastly, since MT-Ins only selectively instruments the instructions inside L_m with a probability of at most P_{m0}, we avoid over-emphasizing thread-interleaving induced transitions, as well as seek a balance between runtime instrumentation overhead and coverage feedback.

4.4.3 Schedule Intervention Instrumentation

Without specifying any scheduling policy or priority, the operating systems usually determine the actual schedule dynamically [91, 92]. As mentioned in §4.2.4, without scheduling intervention the calibration execution during fuzzing may not exhibit diverse interleavings, which is unsatisfactory for covering new transitions.

POSIX compliant systems such as Linux usually provide APIs to control the low level process or thread scheduling [91, 92]. In order to intervene the interleavings during the execution of the multithreaded program, we adjust the thread priorities. This heavily resorts to the POSIX API *pthread_setschedparam*. Our intervention of scheduling follows two principles: 1) the intervened schedule should be possible to happen in reality without this intervention; 2) the schedule should make the interleavings more diverse. We therefore instrument a function call f_I at the start of the thread routine. This instrumented function does two things:

(1) For each newly mutated seed, it calls *pthread_self* in the entry of F_{fork} to collect the runtime thread IDs. This serves two purposes. First, it marks the entering of the multithreading region in order to inform the fuzzer that the execution trace has covered the thread spawning statements. Second, each collected thread ID will also be associated with a unique number N_{ctx} starting from $1, 2, \ldots$, which will be used to calculate the threading context in §4.4.4.

(2) During the calibration execution § 4.5.2, whenever the thread comes to call f_I, it changes the schedule policy to *SCHED_RR*, and assigns a *ranged random value* to its priority. We make the priority value to be *uniformly distributed random* in order to diversify the combinations among threads.

4.4.4 Threading Context Instrumentation

We also apply an instrumentation to track the threading context, which is used to collect thread relevant context for additional feedback during seed selection. This context is collected at the call site of $F_{ctx} = \{\texttt{TLock}, \texttt{TUnLock}, \texttt{TJoin}\}$, each of which has the form $TC = \langle Loc, N_{ctx} \rangle$. Here Loc is the labeling value of previous deputy instruction (c.f. §4.2.2 and § 4.4.2). N_{ctx} is obtained by getting the value of the key identified by the current thread ID from the thread ID map collected at §4.4.3. For each function in F_{ctx}, we keep a sequence of context $\langle TC_1(f), \ldots, TC_n(f) \rangle$,$f \in F_{ctx}$; and at the end of each execution we calculate a hash value $H(f)$ for each of them. The tuple $S_{ctx} = \big\langle H(\texttt{TLock}), H(\texttt{TUnLock}), H(\texttt{TJoin}) \big\rangle$ is a *context-signature* that determines the overall threading context of a specific execution. Essentially, this is a sampling on thread relevant APIs to track the thread information and transitions of a particular run. As we shall see later, this plays an important role in seed selection (§4.5.1).

4.5 Dynamic Fuzzing

The dynamic fuzzing loop follows a typical GBF procedure as described in Algorithm1. We improve the procedures of seed selection (§4.5.1), calibration execution (§4.5.2) and seed triaging (§4.5.3) to fit especially for fuzzing multithreaded programs, based on the instrumentation information provided in §4.4.

4.5.1 Seed Selection

Seed selection decides which seeds to be mutated next. In practice, this problem is reduced to whether the next seed t at the queue header will be selected into T_S, which is depicted by Algorithm 6.

During seed selection, in addition to following AFL's strategy by using *has_new_trace(T_S)* to checking whether T_S contains a "favored seed" t covering a new transition (*cov_new_trace(t)*=true), we also check whether there is at least one seed in

Algorithm 6: Algorithm to Select Next Seed

```
input : Seed queue T_S, current seed in queue t
output: whether t will be selected in this round
if has_new_mt_ctx(T_S) OR has_new_trace(T_S) then
    if cov_new_mt_ctx(t) then
        return true;
    else if cov_new_trace(t) then
        return select_with_prob(P_ynt);
    else
        return select_with_prob(P_ynn);
else
    return select_with_prob(P_nnn);
```

T_S that reaches the multithreading code (i.e., *has_new_mt_ctx(T_S)*). If either of the conditions is satisfied, then there are some "interesting" seeds. Specifically, if the seed has a new threading context, the algorithm directly returns true. If it covers a new trace, it is selected with a probability of P_{ynt}; otherwise, the probability is P_{ynn}. Last, if no interesting seeds in T_S are interesting at all, the algorithm selects with a probability of P_{nnn}. Analogous to AFL's selection strategy, we set $P_{ynt} = 0.95$, $P_{ynn} = 0.01$, $P_{nnn} = 0.15$.

To implement *cov_new_mt_ctx(t)*, we track the context of calling a multithreading API in $F_{ctx} = \{\texttt{TJoin}, \texttt{TLock}, \texttt{TUnLock}\}$ (c.f. §4.4.4) and check whether the context-signature S_{ctx} has been met before — when S_{ctx} is new, *cov_new_mt_ctx(t)* $= true$; otherwise, *cov_new_mt_ctx(t)* $= false$. Note that *cov_new_trace(t)=true* does not imply *cov_new_mt_ctx(t)=true*. The reason is that (1) we cannot instrument inside the body of thread API functions in F_{ctx}, hence *cov_new_trace* cannot track the transitions; (2) *cov_new_mt_ctx* also depends on the thread IDs that *cov_new_trace* is unaware of.

4.5.2 Calibration Execution

Multithreaded programs introduce non-deterministic behaviors when different interleavings are involved. As seen in Algorithm 1, for a mutated seed, a GBF typically executes the target program with multiple times. Owing to f_I, we are now able to tell whether the current non-deterministic behavior is relevant to a multithreading execution. In fact, since we focus on multithreading only, based on the times of the threading forking operations the fuzzer has observed, the fuzzer can distinguish the calibration

on those seeds that have non-deterministic behaviors purely by checking whether the execution traces have touched the threading relevant regions. Further, if we know that a seed may have more distinct values of S_{ctx} (the number of distinct values for seed t is denoted as C_m (t)), there will be more interleavings. To determine the calibration times $\mathbb{N}_c$, we rely on C_m. In AFL, the calibration times is calculated by:

$$\mathbb{N}_c(t) = N_0 + N_v \cdot B_v \tag{4.4}$$

where N_0 is the initial calibration times, N_v is a constant as the "bonus" calibration times for non-deterministic runs, and $B_v \in \{0, 1\}$. B_v=0 if none of the N_0 calibration runs show non-deterministic behaviors, otherwise B_v=1. We argue this to specially fit for multithreading setting.

$$\mathbb{N}_c(t) = N_0 + \texttt{min}\{N_v, N_0 \cdot C_m(t)\} \tag{4.5}$$

In both AFL and DOUBLADE, $N_0 = 8$, $N_v = 32$. For all the $\mathbb{N}_c$ calibration runs, we track their execution traces and count how many different traces (T_{cal}) it exhibits.

4.5.3 Seed Triaging

This handles how to categorize the seeds after calibration execution. DOUBLADE still follows the workflow at Lines 9∼12 in Algorithm 1: if there is a crash, we mark these seeds as vulnerable; if the seed covers new transitions, we append it to the queue. One major difference is that these "seemingly normal" seeds may actually have concurrent bugs. These will be handled in the replaying procedure (§ 4.6.2) after fuzzing. To provide a hint on how many times to be executed on a specific seed during replaying, we keep T_{cal} statistics.

TABLE 4.1: Static information of the 12 evaluated programs.

ID	Project	MT Type	Command Line Options	Binary Size	T_{pp}	N_B	N_I	N_{ii}	$\frac{N_{ii}-N_B}{N_B}$
lbzip2-c	lbzip2-2.5	native	lbzip2 -k -t -9 -z -f -n4 FILE	377k	7.1s	4010	24085	6208	54.8%
pbzip2-c	pbzip2-v1.1.13	native	pbzip2 -f -k -p4 -S16 -z FILE	312k	0.9s	2030	8345	2151	6.0%
pbzip2-d	pbzip2-v1.1.13	native	pbzip2 -f -k -p4 -S16 -d FILE	312k	0.9s	2030	8345	2151	6.0%
pigz-c	pigz-2.4	native	pigz -p 4 -c -b 32 FILE	117k	5.0s	3614	21022	5418	49.9%
pxz-c	pxz-4.999.9beta	OpenMP	pxz -c -k -T 4 -q -f -9 FILE	42k	1.2s	3907	30205	7877	101.6%
xz-c	XZ-5.3.1alpha	native	xz -9 -k -T 4 -f FILE	182k	8.4s	4892	34716	8948	82.9%
gm-cnvt	GraphicsMagick-1.4	OpenMP	gm convert -limit threads 4 FILE out.bmp	7.6M	224.4s	63539	383582	98580	55.1%
im-cnvt	ImageMagick-7.0.8-7	OpenMP	convert -limit thread 4 FILE out.bmp	19.4M	434.2s	128359	778631	200108	55.9%
cwebp	libwebp-1.0.2	native	cwebp -mt FILE -o out.webp	1.8M	56.3s	12117	134824	33112	173.3%
vpxdec	libvpx-v1.3.0-5589	native	vpxdec -t 4 -o out.y4m FILE	3.8M	431.6s	31638	368879	93400	195.2%
x264	x264-0.157.2966	native	x264 –threads=4 -o out.264 FILE	6.4M	1701.0s	38912	410453	103926	167.1%
x265	x265-3.0_Au+3	native	x265 –input FILE –pools 4 -F 2 -o	9.7M	78.3s	22992	412555	89408	288.9%

TABLE 4.2: Fuzzing and Replaying Results on AFL, DOU-AFL, and DOUBLADE

ID		Gen Tests			Vuls			C-Bugs
		N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_c	V_m	V_s	
DOUBLADE	**lbzip2-c**	3350	753	22.5%	0	0	0	1
	pbzip2-c	175	50	28.6%	0	0	0	0
	pbzip2-d	926	7	0.8%	0	0	0	0
	pigz-c	737	592	80.3%	0	0	0	1
	pxz-c	2305	958	41.6%	0	0	0	0
	xz-c	1327	257	19.4%	0	0	0	0
	gm-cnvt	3921	1872	47.7%	0	0	0	3
	im-cnvt	3289	2460	74.8%	8	2	1	2
	cwebp	4755	2234	47.0%	9	0	1	0
	vpxdec	13046	3117	23.9%	345	1	2	1
	x264	4465	4252	95.2%	3	1	0	4
	x265	6420	4701	73.2%	49	0	1	0
DOU-AFL	**lbzip2-c**	4155	1352	32.5%	0	0	0	1
	pbzip2-c	227	59	26.0%	6	1	0	0
	pbzip2-d	982	11	1.2%	0	0	0	0
	pigz-c	921	789	85.7%	0	0	0	1
	pxz-c	2804	1675	59.7%	0	0	0	0
	xz-c	1359	268	19.7%	0	0	0	0
	gm-cnvt	4677	3265	69.8%	0	0	0	5
	im-cnvt	4036	2092	51.8%	0	0	0	4
	cwebp	5549	3095	55.8%	11	0	1	0
	vpxdec	13927	3302	23.7%	364	1	2	1
	x264	4457	4197	94.2%	6	1	0	6
	x265	6743	5007	74.3%	43	0	1	0
AFL	**lbzip2-c**	4503	1637	36.4%	0	0	0	1
	pbzip2-c	253	78	30.8%	4	1	0	0
	pbzip2-d	1231	47	3.8%	5	1	0	0
	pigz-c	921	789	85.7%	0	0	0	1
	pxz-c	3658	2523	69.0%	0	0	0	0
	xz-c	1598	493	30.9%	0	0	0	0
	gm-cnvt	4677	3265	69.8%	0	0	0	5
	im-cnvt	4355	3671	84.3%	21	4	1	3
	cwebp	5701	3347	58.7%	8	0	1	0
	vpxdec	14665	3656	24.9%	336	2	2	3
	x264	5023	4832	96.2%	7	1	0	9
	x265	6433	5012	78.0%	29	0	1	0

4.6 Evaluation

We develop DOUBLADE upon FOT and SVF [65, 93, 94]. The static instrumentation component utilizes the LLVM analysis while the thread-aware callgraph construction leverages SVF's inter-procedural value-flow analysis. In total this static part has around 3000 lines of C/C++ code. Also, we modify FOT's fuzzing engine with about 800 lines of C code. And the replay part takes additional 1500 lines of Python code. We archive all the supporting materials at [98]. These include the initial fuzzing seeds, the source

code, and the findings of our evaluation.

Specifically, we design the following research questions to conduct our evaluation:

RQ1 Can DOUBLADE generate seeds that trigger more non-deterministic behaviors?

RQ2 What is the capability of DOUBLADE in detecting vulnerabilities?

RQ3 Can the generated seeds help bug detectors to find more concurrency faults?

4.6.1 Evaluation Setup

Our experiments are conducted on an Intel(R) Xeon(R) Platinum 8151 CPU @ 3.40GHz with 28 cores, running a 64-bit Ubuntu 18.04 LTS system; during experiments, we use 24 cores and retain 4 cores for other processes.

4.6.1.1 Settings of the fuzzers

We set up three fuzzers for evaluation:

1) **AFL** is the current state-of-the-art GBF with AFL-Ins instrumentation and the thread-unaware fuzzing strategies. To the best of our knowledge, there are no other open-source fuzzers that target multithreaded programs. Meanwhile, evolved AFL shows comparable performance with recent techniques [69, 90]. Therefore, we use AFL as the baseline fuzzer.

2) **DOU-AFL** enhances AFL with schedule-intervention (§4.4.3) and threading-context instrumentations (§4.4.4), as well as all the dynamic strategies (§4.5) .

3) **DOUBLADE** is our proposed fuzzer. It has all the static and dynamic strategies proposed in §4.4 and §4.5. DOUBLADE differs from DOU-AFL only on exploration-oriented instrumentation — the latter uses AFL-Ins.

During the evaluation, all the fuzzers are executed in the "fidgety" mode as suggested in [70]. We also disable AFL's CPU affinity binding strategy. This is because

that the threads in multithreading pro- grams are intended to be mapped in different cores and execute in parallel in order to boost overall performance; setting the CPU affinity will make all these threads bound to one CPU core. In practice, setting CPU affinity frequently makes the fuzzing procedure fail due to heavy loads.

4.6.1.2 Statistics of the evaluation dataset

The dataset for evaluation consists of the following real-world software projects.

1) Parallel compress/decompress utilities including **pigz**, **lbzip2**, **pbzip2**, **xz**, and **pxz**. These tools have been present in GNU/Linux distributions for many years and are integrated into the widely used GNU tar utility.

2) **ImageMagick** and **GraphicsMagick** are two widely used soft- ware suites to display, convert, edit images files.

3) **libvpx** and **libwebp** are two WebM projects. They are used by mainstream browsers like Google Chrome, Firefox, and Opera.

4) **x264** and **x265** are two most established video encoders for H.264/AVC and HEVC/H.265 formats respectively.

All these software systems have been intensively tested by AFL, LibFuzzer, or added into Google's OSS-Fuzz project. We try to use their *latest* versions at the time of evaluation; the only exception is libvpx, which we use version v1.3.0-5589 to reproduce vulnerabilities and concurrency bugs.

Table 4.1 lists the statistical data of the benchmarks. The first two columns show the benchmark names and their host software projects. The third column represents the multithreading implementation — either the OpenMP [99] APIs or the standard pthread library. The next column specifies the command line options.

The rest columns state the static instrumentation data. The "Binary Size" column calculates the sizes of the instrumented binaries. Column T_{pp} records the preprocessing time (c.f. §4.4.1). The program *vpxdec* cost the most time — about 30 minutes. The

last three columns N_B, N_I, and N_{ii} depict the number of basicblocks, the number of total instructions, and the number of included instructions for DOUBLADE instrumentation (c.f. §4.4.2), respectively. Recall that AFL-Ins does the instrumentation over the *deputy* instruction of each basicblock, so N_B also implicates the number of instrumented instructions by AFL. The last column is the ratio of more instructions DOUBLADE instrumented compared to AFL.

4.6.2 Overall Results

We run each aforementioned fuzzer 4 times against all the 12 benchmark programs, with the time threshold as 6 hours. Table 4.2 shows the overall evaluation results in terms of generated seeds, the detected vulnerabilities, and the found concurrency bugs. The *underlined* table entries indicate they have the best performance among all the 3 evaluated fuzzers.

Seed Generation. We collect both the number of all new tests (N_{all}) and the number of tests that are multithreading-relevant (N_{mt}). We sum up the generated seeds of all 4 fuzzing runs to form N_{all} and N_{mt} in Table 4.2. The $\frac{N_{mt}}{N_{all}}$ column shows the percentage of N_{mt} against N_{all}.

Vulnerability Detection. We denote the total number of detected crashes as N_c. For each unique crash, we manually triage it to corresponding vulnerability type based on its root causes. Further, we group the detected vulnerabilities into two categories: the vulnerabilities triggered in multithreading, whose number is denoted as V_m; the vulnerabilities triggered in single-threading, whose number is V_s.

Concurrency Bugs. We detect concurrency bugs in the *replaying procedure*. Specifically, we compile the target programs with ThreadSanitizer [100] and re-run them with the new seeds generated during Seed Generation. Also, we design a *weighted round-robin* strategy to continuously replay the seeds against the non-deterministic program behaviors until reaching a given time budget (2 hours).

Since ThreadSanitizer only needs several seconds to execute a seed, our *replaying procedure* could finish thousands of rounds of seed replay. In each round, the running

chances for AFL seeds remain 1 (referred to ❶) while the chance of each DOUBLADE and DOU-AFL seed is T_{cal} (referred to ❷). Hence, ThreadSanitizer can schedule AFL seeds as evenly as possible while weighted seeds from DOUBLADE are treated with optimal schedules.

We inspect the reported concurrency bugs and categorize them based on the trace information. The column C-Bug depicts the final number of bugs.

4.6.3 Seed Generation (RQ1)

In Table 4.2 we observe that DOUBLADE surpasses both DOU-AFL and AFL in the number of generated tests.

DOUBLADE exhibits superiority in generating seeds that exhibit non-deterministic program behaviors — in almost all the benchmarks DOUBLADE generates the most seeds. For *pbzip2-d* DOUBLADE generated 47 seeds, which is $6.7x$ the number of AFL's seeds (totally 7) and $4.3x$ the number of DOU-AFL's seeds (totally 11); The only exception is *x265* where DOU-AFL generates more valuable seeds than DOUBLADE.

The percentages of DOUBLADE's seeds against all the generated seeds are also impressive — DOUBLADE wins performance comparison over all the benchmarks. Regarding program *pbzip2-d*, DOUBLADE's result of ratio $\frac{N_{mt}}{N_{all}}$ is much high than AFL and DOU-AFL (3.8% vs 0.8% vs 1.2%). For the benchmark where AFL has already achieved decent result, e.g., 95.2% for *x264*, DOUBLADE can even improve it to 96.2%.

In addition, DOU-AFL also outperforms AFL on N_{all}, N_{mt}, and $\frac{N_{mt}}{N_{all}}$ in most programs. Considering DOU-AFL uses two instrumentation and the fuzzing strategy from DOUBLADE, we can conclude that our proposed schedule intervention and thread-aware instrumentation work best for seed generation.

Answer to RQ1: DOUBLADE has *overwhelming* advantages in increasing the number and percentages of multithreading relevant seeds for multithreaded programs. The proposed three categories of instrumentation and fuzzing strategies significantly benefit the seed generation.

4.6.4 Vulnerability Detection (RQ2)

We refer to the N_c, V_m and V_s columns to evaluate DOUBLADE's vulnerability detection capability. We intend to detect more concurrency relevant vulnerabilities (V_m). As shown in Table 4.2, DOUBLADE detects the most crashes in all programs. In total, DOUBLADE detects 9 such vulnerabilities, while DOU-AFL and AFL only detect 5 and 4 respectively. These vulnerabilities can be further divided into three groups.

Concurrency-bug Induced Vulnerabilities (V_{cb}). The 4 vulnerabilities found in *im-cnvt* all belong to this group. The root causes are the misuses of caches shared among threads, which causes the data races. The generated seeds exhibit various symptoms such as use after free, double free, heap buffer overflow, and invalid memory read. Subsequently, the execution traces of the crashes also differ.

V_m that are triggered with multi-thread only but not induced by concurrency-bugs. For instance, the crash in *pbzip2-d* stems from a stack overflow error in the processing of decompressing a corrupted BZip2 file. The root cause is that the call to *pthread_attr_setstacksize* constrains the stack size of each thread. But certain corrupted BZip2 files may use up the stack in a short time. This crash can never happen when *pbzip2-d* works in single-threaded mode since *pbzip2-d* executes another *separate function* when only one thread is specified. In fact, the vulnerability can only be triggered when the carefully crafted input is fed to multithreaded *pbzip2-d*. In our evaluation, DOUBLADE detects this crash in 3 runs (out of 4) with 5 proof-of-crash generated seeds; while AFL or DOU-AFL cannot detect it.

We speculate the reason that DOUBLADE generate more valuable seeds (totally 47) than other two fuzzers (AFL: 7; DOU-AFL: 12). To support this conjecture, we feed both AFL and DOU-AFL the 47 DOUBLADE seeds — then both AFL and DOU-AFL are able to trigger the crash in 40 minutes.

V_m that can be triggered with single-thread. This refers to the crashes in benchmarks *vpxdec* and *x264*. DOUBLADE detects 2 vulnerabilities in *vpxdec* while both AFL and DOU-AFL find 1. And all three fuzzers report 1 vulnerability in *x264*.

Apart from multithreading relevant vulnerabilities (V_m), we indeed find several vulnerabilities beyond the concurrency context of the benchmarks (named as **ST-Vul**). In column V_s, it is observable that *all* three fuzzers detect *an equal number* of such vulnerabilities. This also implicates that fuzzing is good at finding "shallow bugs" [101] even with less coverage feedback.

The developers have confirmed all the 10 newly discovered vulnerabilities in Table 4.2 (the vulnerabilities inside *vpxdec* no longer exists in its latest version.), and 2 CVEs have been assigned.

Answer to RQ2: DOUBLADE can detect more vulnerabilities than state-of-the-art fuzzers. It demonstrates *superiority* in detecting *multithreading-relevant* vulnerabilities while retains *equivalent capability* in detecting other vulnerabilities.

4.6.5 Concurrency Bug Detection (RQ3)

We inspect the C-Bug column to assess the capability of bug detection and find that DOUBLADE reports the most number of bugs in *all* buggy benchmarks. Based on DOUBLADE's seeds, ThreadSanitizer successfully detected the thread leaks in *lbzip2-c* and potential deadlocks in *pigz*. For the other projects (*gm-cnvt*, *im-cnvt*, *vpxdec* and *x264*), DOUBLADE detected more concurrency bugs – coincidentally, these are *all* data races.

Case Study on *gm-cnvt*. In this benchmark, ThreadSanitizer detected 5, 4, and 3 data races by replaying seeds generated by DOUBLADE, DOU-AFL, and AFL, respectively. from the parallel image converting — read and write operations of a shared variable row_count can happen simultaneously. Compared with DOU-AFL, the "quality" of DOUBLADE seeds tend to be better. To confirm that the weight T_{cal} indeed helps bug detection, we replay 2092 DOU-AFL seeds with the two strategies ❶ and ❷ (c.f. §4.6.2). We repeat each replay 6 times. When a replay process has detected *all* the 4 categories of "ground truth" concurrency bugs (DOU-AFL detected C-Bug on *gm_cnvt* in Table 4.2), we record the used time in *minutes* in Table 4.3.

We analyze the results in Table 4.3. Compared to ❶, we see that ❷ reduces the average time-to-exposure from 66.5 minutes to 34.1 minutes. Moreover, ❷ is more

TABLE 4.3: Time-to-Exposure of all ground truth concurrency bugs during 6 replay procedures with strategies ❶ and ❷.

	#1	#2	#3	#4	#5	#6	Avg	Variance
❶	55.3	92.1	21.8	93.7	101.5	34.7	66.5	959.2
❷	33.4	52.2	33.5	37.6	24.7	23.3	34.1	91.0

stable since the timing variance is much smaller than that of ❶. This convinces us that given a set of seeds ❷ indeed exposes the bugs faster and more stably. Since ❷ is relevant to schedule-intervention instrumentation (§ 4.4.3) and calibration execution (§ 4.5.2), this also indicates that the strategy helps for concurrency bug identification.

We have reported the detected bugs to their project maintainers. At the time of writing, GraphicsMagick, and x264 developers have confirmed the reported issues.

Answer to RQ3: DOUBLADE generated seeds are useful for bug detectors to detect concurrency bugs; T_{cal} additionally helps reveal concurrency bugs earlier.

4.6.6 Discussion

Our approach to fuzzing multithreaded programs is relying on fuzzing to detect all the crashes regardless of their root causes, and then using ThreadSanitizer to run against the generated seeds to find more concurrency bugs. Here, we discuss some alternatives.

4.6.6.1 Enforce single-thread run during fuzzing to detect vulnerabilities.

Indeed fuzzing single-threaded programs can reduce the non-deterministic behaviors, which probably benefits the fuzzing efficiency. However, the problem is that some programs are designed to have some threading features such as thread-pools (e.g. *x265*) that there is no easy way to enforce the programs to run in single-threading mode with command line options. On the other hand, the generated seeds will be *much less useful* for ThreadSanitizer's concurrency bug detection. The rationale is that the fuzzing procedure keeps seeds purely based on coverage, it may generate large numbers of seeds that are irrelevant to multithreading. Worse still, we may encounter the scenarios similar

to *pbzip2-d*: the program runs single-threading and multithreading decompression with two *separate implementations*.

4.6.6.2 Integrate concurrency bug detection directly during fuzzing.

This approach seems straightforward however is not quite practical. The reason is that concurrency bug detection techniques brings considerate overhead: even the most widely used ThreadSanitizer still claims to have $5x \sim 15x$ overhead compared to the regularly compiled programs [100, 102]. On the other hand, GBFs usually generate large numbers of seeds that the bottleneck becomes the execution on the ThreadSanitizer instrumented programs. In fact, in our empirical study on lbzip2's [103] parallel decompression functionality with AFL, after running 20 hours, a ThreadSanitizer instrumented binary had an average speed of 2.57 executions per second, with only 18 paths explored as decided by AFL; while the binary without ThreadSanitizer could run as fast as 38.9 executions per second, with more than 10000 paths explored. The worst thing is that if ThreadSanitizer does not report any bugs in one execution, it cannot in turn provide any feedback that guides the mutation or seed selection. It is however may be possible to integrate other exhaustive concurrency bug detectors such as UFO [104] to apply classifications to avoid inefficient mutations on the generated seeds.

4.7 Related Work

The work of this Chapter is relevant to the following research aspects.

4.7.1 Static Concurrency Bug Prediction

Static concurrency bug predictors aim to approximate the runtime behaviors of a concurrent program without actual execution. Several static approaches have been proposed for analyzing pthread and Java programs [94, 105–108]. LOCKSMITH [105] uses existential types to correlate locks and data in dynamic heap structures for race detection.

Goblint [106] relies on a thread-modular constant propagation and points-to analysis for detecting concurrent bugs by considering conditional locking schemes. FSAM [94] proposes a sparse flow-sensitive pointer analysis for pthread C programs using context-sensitive thread interleaving analysis. D4 [107] presents a real-time concurrency bug detection approach using parallel and incremental pointer analysis. Currently DOUBLADE relies on lightweight static results to guide GBFs to perform three categories of instrumentation for efficient fuzzing. We envision that DOUBLADE can benefit more by integrating insights from these aforementioned static bug prediction techniques.

4.7.2 Dynamic Analysis on Concurrency Bugs

There also exists a large body of dynamic analyzers. Essentially these can be divided into two categories: the concurrency bug detection techniques and the strategies to trigger violation conditions that can be captured by these detection techniques.

Dynamic race detectors [102, 109–112] typically instrument target programs and monitor the memory and synchronization events [89]. The two fundamentals are *happens-before model* [109] and *lockset models* [110]; the former reports a race condition when two threads access a shared memory location and the accesses are causally unordered, and the latter considers a potential race if two threads access a shared memory location without locking. Modern detectors such as ThreadSanitizer [102], Valgrind [113] usually apply a hybrid strategy to combine these two models for accuracy and efficiency. Our work does not aim to improve the existing dynamic detection techniques. In fact, we rely on ThreadSanitizer to detect concurrency bugs.

The other dynamic analyses focus on how to trigger the concurrency violation conditions. These include the random testings that mimic non-deterministic program executions [114–118], the regression testing methods [119, 120] that targets interleavings from code changes, the model checking [121–124] and hybrid constraint solving [104, 125, 126] approaches that systematically checks or execute possible thread schedules, or heuristically avoid fruitless executions [127, 128]. Our work differs from all of them in the sense that our goal is to cover more execution paths by generating different seeds that exercise different paths in multithreading-relevant settings. Indeed,

we also provide the schedule-intervention instrumentation. However, the main goal is to diversify the schedule introduced interleavings (i.e., transitions).

4.7.3 Dynamic Fuzzing Techniques

Many techniques are proposed to improve the effectiveness of exposing vulnerabilities.

For general-purpose fuzzing [15, 17–19, 22, 37], the most common strategies are to utilize more feedbacks for execution path explorations. Steelix [22] applies a lightweight binary transformation to keep track of program states during magic byte comparisons, which helps guide where and how the mutations will be processed. Angora [19] distinguishes different calling context when calculating deputy instruction transitions and is able to keep more valuable seeds. DOUBLADE inspires from these two techniques in the sense that it aims to provide more valuable feedback for multithreading environment: the stratified coverage-oriented instrumentation on suspicious interleaving scope L_m bookkeeps more transitions for multithreading part and the thread-context instrumentation additionally differentiates execution context of various threads.

Other fuzzing techniques focus on detecting certain categories of bugs or vulnerabilities [1, 16, 21, 23, 129–131]. DOUBLADE focuses on the concurrently executed segments and aims to reveal more vulnerabilities and bugs in a multithreaded environment. Therefore, our strategies for seed selection, seed scheduling are all tuned for this purpose.

There are some fuzzing works for multithreading induced vulnerabilities. RAZZER [132] uses the deterministic thread interleaving technique implemented at the hypervisor and generates seeds that can trigger data races in Linux kernel. This is a kernel fuzzer whose generated seeds correspond to sequences of syscalls; while DOUBLADE mutates the input file content of an application program. Liu et al [133] is the most relevant to DOUBLADE in the sense that it is also a file fuzzer that works for multithreaded user-space programs. One major difference is their work only detects concurrency bug induced vulnerabilities. In terms of methodologies, their work relies on heavyweight static analysis from LOCKSMITH [105] therefore it faces scalability

issues. On the other hand, we propose our novel exploration-oriented instrumentation and the comprehensive strategies to generate more seeds relevant to multithreading. The approach is both scalable and effective.

4.8 Conclusion

We presented DOUBLADE, a novel technique that empowers thread-aware seed generation to GBF for fuzzing multithreaded programs. Our approach performs three categories of instrumentation guided by static analysis to explore new program paths that are relevant to thread-interleavings. Based on the additional feedback provided by these instrumentation, we apply a series of dynamic strategies optimized for exercising thread-interleaving related paths and generating multithreading relevant seeds. Experiments on 12 real-world programs demonstrate that DOUBLADE significantly outperforms the state-of-the-art grep-box fuzzer AFL in generating more valuable seeds, detecting more multithreading relevant vulnerabilities and concurrency bugs.

Chapter 5

The FOT Fuzzing Framework

5.1 Introduction and Motivation

In spite of the popularity and effectiveness of applying greybox fuzzing techniques in detecting vulnerabilities(c.f. Sec 2.1), there does not exist a general greybox fuzzing framework to reuse, integrate and evaluate fuzzing extensions as well as try new techniques. For example, AFL's core fuzzing logic, around 8000 lines of code, all resides in one single file, with more than 100 global variables. Therefore, the integration of one new feature often involves modifications in multiple places; these changes are usually erroneous since a minor mistake may cause whole logic wrong. In reality, AFL is highly coupled as it is designed with limited options for configurations [27]. Further, most of the existing fuzzers are extensible to integrate with new features. Hence, a fuzzing framework is preferred to allow for both easy *configuration* and *extension*.

In this sense, we propose our fuzzing framework, *Fuzzing Orchestration Toolkit* (FOT), which has the following three properties.

(1) **Versatility.** FOT provides a fuzzing toolkit, including static analysis enhanced fuzzing preprocessors, collaborative fuzzing core, as well as a set of toolchains for additional analysis purpose.

(2) **Configurability.** FOT has builtin support for multiple configuration options, which makes it easy to tweak fuzzing parameters in order to improve the overall fuzzing effectiveness.

(3) **Extensibility.** FOT consists of the library for general-purpose fuzzing utilities, as well as the miscellaneous tools on top of it, which makes it possible for developers to customize their own fuzzers with modest efforts.

5.2 Architecture

This section describes the key components of FOT. Figure 5.1 depicts the overall architecture of FOT. It has three parts (labeled in pink): the *preprocessor*, the *fuzzer*, and the *complementary toolchains*. The components are labeled in *blue* while the inputs and inputs/outputs are in *grey*.

5.2.1 Preprocessor

This part consists of multiple tools to collect statically calculated metrics which will be encoded as instrumentation stubs inside the target programs.

5.2.1.1 Static Analyzer

This includes several analyzers to extract interesting semantics from the target program, such as control flow graph generation, data flow analysis results, as well as the metric utility to fit them for instrumentation [134, 135]. It is *configurable* to generate different levels of static information. It is *extensible* as developers are allowed to add new types of static analysis as long as the generated result follows the specified format.

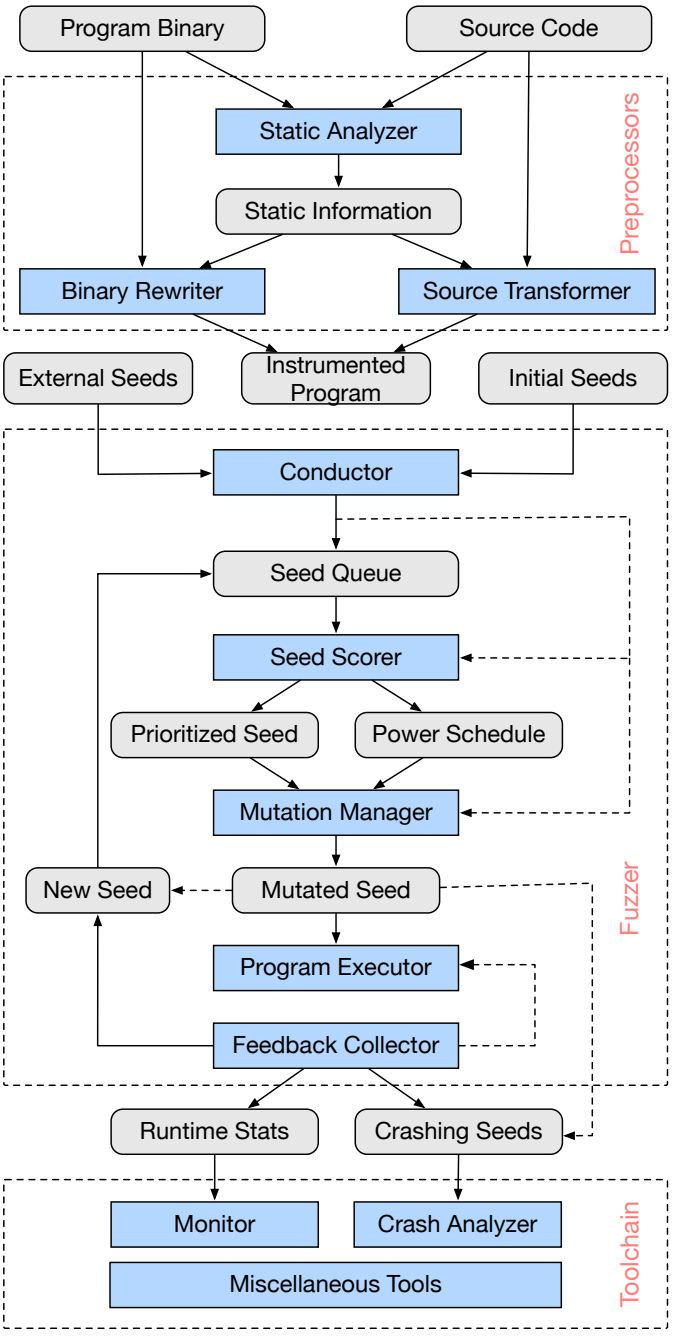

FIGURE 5.1: The architecture of the FOT fuzzing framework

5.2.1.2 Instrumentor

Binary rewriter and *source transformer* instrument static results generated by the static analyzer into the target program for the fuzzer to collect feedback during runtime execution. FOT supports LLVM based transformation when the source code is available, and Dyninst [136] rewriting when only the binary is provided. It is *configurable* since it allows instrumentation either on source code or binary. It is *extensible* since developers

can use other instrumentors (e.g., Intel PIN [36]), as long as the instrumented code can embed the static information and provide sufficient feedback during fuzzing.

5.2.2 Fuzzer

This part deals with the actual fuzzing. Basically, the core of fuzzing is a loop that continuously picks seeds from a seed queue, applies mutations on the seeds, executes the target program against seed variants, and collects statistics for subsequent iterations.

5.2.2.1 Conductor

FOT a *conductor* to schedule workloads of different workers. This specifically allows for parallel fuzzing in a multithreading mode by collaborating different fuzzing instances. In addition, a monitoring worker can also be waken up regularly to retrieve seed inputs from external tools, including symbolic executors (e.g., KLEE [32]), random seed generators (e.g., Radamsa [137]), by monitoring a specific seed directory. It is *configurable* since end-users are allowed to choose different fuzzing modes. It is *extensible* since it can interoperate with other external tools.

5.2.2.2 Seed Scorer

The seed scorer is in charge of selecting a seed from the queue for mutation (seed selection) and determining how many new seeds are to be generated from the selected seed (power scheduling). It is *configurable* since end-users can choose builtin scoring strategies to evaluate seeds. It is *extensible* since developers can implement their own strategies with the provided interfaces.

5.2.2.3 Mutation Manager

The mutation manager schedules different kinds of mutators, which includes bitwise flips, bytewise flips, arithmetic operations on bytes, byte chunk addition, modification or deletion, crossover between two seeds, etc. It is *configurable* as FOT has various

mutators (and their combinations) for end-users to choose from. It is *extensible* as developers can implement their own mutators with the provided interfaces.

5.2.2.4 Program Executor

The program executor drives the execution of the target program. It is *configurable* since the default executor in FOT allows end-users to choose whether or not to use forkserver [27] during fuzzing. It is *extensible* since developers can extend the program executor for various scenarios. For example, they may add a secondary executor to execute another program for differential testing.

5.2.2.5 Feedback Collector

The feedback collector collects the runtime feedback based on the instrumented information (e.g., path coverage) as well as the program's execution status. It is *configurable* as end-users are allowed to select from the builtin feedback options. For now, the feedback can be at the basicblock level (same as AFL) or function level (specific to FOT). It is *extensible* as end-users can specify their customized types of feedback for collection.

5.2.3 Complementary Toolchains

FOT contains various complementary tools which help to make the framework *versatile*. For instance, we implemented a web-based frontend user interface to observe the overall results, which provides intuitive and fruitful information for manual monitoring. We also have a crash analyzer to analyze the detected crashes and automatically generate reports [138]. This greatly reduces the manual efforts of crash triaging. Many other tools are also being added to complement the fuzzer.

5.2.4 Implementation

The FOT project started in June 2017 and has been actively developed by two researchers. It is implemented with 16000 lines of Rust for core fuzzing modules, together

with 4500 lines of C/C++ for the preprocessor, 4200 lines of Java for structure-aware mutation, and 2400 lines of Python for the complementary toolchain.

5.3 Relations with Other Fuzzing Tools

Table 5.1 compares FOT with existing fuzzing frameworks with respect to 10 fuzzing features. As we can see, the existing fuzzing frameworks, AFL, libFuzzer, and honggfuzz, lack FOT's features in various aspects. FOT stands out in the sense that it provides multiple configurations for advanced end-users; it is also highly modularized, suitable to be extended with other fuzzing techniques.

Most current fuzzing extensions can be easily integrated into FOT owing to its design. In reality, they can be applied with some extensions to the different components in Figure 5.1 and can be used together with the configuration interface.

1) HAWKEYE is implemented by applying target location-specific static analysis, as well as optimizations on seed selection, power scheduling, mutations.

2) DOUBLADE is implemented by applying thread-aware static analysis and special instrumentations, as well as the optimizations on seed selection, power scheduling strategies, program executor, feedback collector, etc.

3) AFLFast [15] can be implemented by applying a Markov Chain model-based seed *power scheduling* in the fuzzer.

4) AFLGo [16] can be implemented by a combination of *static analyzer*, *instrumentation* and *power scheduling*.

5) CollAFL [18] can be implemented by using a collision-resistant algorithm to increase the uniqueness of the path trace labeling during *instrumentation*.

6) Skyfire [21], Radamsa [137], Csmith [139] can be used to generate *external seeds* which will be *imported*, or integrated as structure-aware mutators conducted by *mutation manager*. Symbolic executors such as KLEE [32] can be integrated in the Driller's [101] style with the help of *conductor*.

TABLE 5.1: Comparisons between fuzzing frameworks (○: not supported, ◐: partially supported, ●: fully supported)

Features \ Framework	AFL	libFuzzer	honggfuzz	FOT
Binary-Fuzzing Support	●	○	●	●
Multi-threading Mode	○	●	●	●
In-memory Fuzzing	●	●	●	●
Advanced Configuration	○	◐	○	●
Modularized Functionality	○	◐	○	●
Structure-aware Mutation	○	○	○	◐
Interoperability	○	○	○	◐
Toolchain Support	●	○	○	●
Precise Crash Analysis	○	○	●	●
Runtime Visualization	◐	○	○	●

5.3.1 Newly Detected Vulnerabilities

FOT has been used to fuzz more than 100 widely used open source projects and it has detected more than 200 vulnerabilities, among these 51 CVE IDs have been assigned. The detailed information is available in Appendix B. Notably, FOT has detected multiple vulnerabilities with *high* or *critical* severity. For example, CVE-2019-9169 [140] stresses a vulnerability in the GNU C Library (a.k.a. glibc or libc6 on Linux distributions) through 2.29, where the function *proceed_next_node* inside POSIX regular expression (regex) component has a heap-based buffer-over-read via an attempted case-insensitive regex match. Thanks to the interoperability with the regex generators, FOT is able to generate more meaningful seeds that cover more corner cases of the regex components. According to CVSS v3.0 [141], it is scored 9.8 out of 10.0 (critical severity); according to CVSS v2.0 [142], it has a score of 7.5 (high severity). As a comparison, the notorious HeartBleed (CVE-2014-0160) is scored 5.0 medium severity according to CVSS v2.0 (no CVSS v3.0 for it). Many of other vulnerabilities discovered by FOT also have high severity, e.g, CVE-2018-15822, CVE-2018-14394, CVE-2018-14395 in FFmpeg, CVE-2018-19837, CVE-2018-19838, CVE-2018-19839, CVE-2018-20821, CVE-2018-20822 in libsass, CVE-2018-14560, CVE-2018-14561, CVE-2019-11470, CVE-2019-11472 in ImageMagick, and CVE-2019-11473, CVE-2019-11474 in GraphicsMagick. We envision that with more integrated techniques, FOT will have the capabilities to detect more vulnerabilities that are hard to be revealed by existing fuzzers.

Chapter 6

Type Checking Based Verification

6.1 Introduction and Motivation

Owing to the pervasive use of mobile applications (apps), mobile security has become increasingly important for our daily life. Among all different kinds of mobile devices, Android accounts for the majority of them. Due to this, security analysis on them has been of significant interest. There has been a number of analyses on Android security ([12, 143–147]) that focus on detecting potential security violations. Here we are interested in the problem of designing secure apps. In particular, we aim to provide a guarantee for information flow security in the constructed apps. Although testing approaches are widely used in detecting Android security issues such as malware analysis, it however fails to stress the "higher level" permission relevant information leakage issue. In fact, testing approaches can only detect some data leakage issues, however cannot guarantee the absence of leakage. In order to achieve this, we apply a type checking based verification approach.

6.1.1 Motivating Examples

During designing an information flow type system for Android, we encounter a common pattern of conditionals that would not be typeable using conventional type systems. Consider the pseudo-code in Listing 6.1. Such a code fragment could be part

```
1 String getContactNo(String name) {
2     String number;
3     if(checkPermission(READ_CONTACT))
4         number = ... ;
5     else number = "";
6     ret number;
7 }
```

LISTING 6.1: Sample code for getting contact info with a permission check.

of a phone dialer or a social network service app such as Facebook, WhatsApp, where *getContactNo* provides a public interface to query the phone number associated with a name. The (implicit) security policy in this context is that contact information (the phone number) can only be released if the calling app has READ_CONTACT permission. The latter is enforced using the *checkPermission* API in Android. Suppose phone numbers are labeled with H, and the empty string is labeled with L. If the interface is invoked by an app that has the required permission, the phone number (H) is returned; otherwise an empty string (L) is returned. In both cases, no data leakage happens: in the former case, the calling app is authorized; and in the latter case, no sensitive data is ever returned. By this informal reasoning, the function complies with the implicit security policy and it should be safe to be called in *any* context, regardless of the permissions the calling app has. However, in the traditional (non-value dependent) typing rule for the if-then-else construct, one would assign *the same* security labels to both branches, and the return value of the function would be assigned level H. As a result, if this function is called from an app with *no* permission, assigning the return value to a variable with security labels L has to be rejected by the type system even though no sensitive information is leaked. To cater for such a scenario, we need to make the security type of *getContactNo depend on* the permissions possessed by the caller.

Banerjee and Naumann [148] proposed a type system (referred to as BN system) that incorporates permissions into function types. Their type system was designed for an access control mechanism different from ours, but the basic principles are still applicable. In BN system, a Java class may be assigned a set of permissions which need to be *explicitly enabled* via an **enable** command for them to have any effect. We say a permission is *disabled* for a class if it is *not assigned* to the class, or it is *assigned* to the class but is *not explicitly enabled*. Depending on the permissions of the calling class (corresponding to an *app* in the above example), a function such as *getContactNo*

can have a collection of types. In BN system, the types of a function take the form $(l_1, \ldots, l_n) \xrightarrow{P} l$ where $l_1, \ldots, l_n$ denote security labels of the input, l denotes the security labels of the output and P denotes a set of permissions that are disabled by the caller. The idea is that permissions are guards to sensitive values. Thus conservatively, one would type the return value of *getContactNo* as L only if one knows that the permission READ_CONTACT is disabled. In BN system, *getContactNo* admits:

$$\textit{getContactNo} : L \xrightarrow{P} L \qquad \textit{getContactNo} : L \xrightarrow{\emptyset} H$$

where $P = \{\text{READ\_CONTACT}\}$. When typing a call to *getContactNo* by an app without permissions, the first type of *getContactNo* is used; otherwise the second type.

In BN system, the typing judgment is parameterized by a permission set Q containing the permissions that are currently known to be disabled. The set Q may or may not contain all disabled permissions. Their language features a command "**ckp**(P) c_1 **else** c_2", which means that if the permissions in the set P are *all enabled*, then the command behaves like c_1; otherwise it behaves like c_2. The typing rules for the **ckp** command (in a much simplified form) are:

$$\text{(R1)}\ \frac{Q \cap P = \emptyset \qquad Q \vdash c_1 : t \qquad Q \vdash c_2 : t}{Q \vdash \textbf{ckp}(P)\ c_1\ \textbf{else}\ c_2 : t}$$

$$\text{(R2)}\ \frac{Q \cap P \neq \emptyset \qquad Q \vdash c_2 : t}{Q \vdash \textbf{ckp}(P)\ c_1\ \textbf{else}\ c_2 : t}$$

where Q is a set of permissions that are disabled. When $Q \cap P \neq \emptyset$, then at least one of the permissions in P is disabled, thus one can determine statically that "**ckp**(P)" would fail and only the *else* branch would be executed at runtime. This case is reflected in the typing rule R2. When $Q \cap P = \emptyset$, there can be two possible runtime scenarios. One scenario is that all permissions in P are enabled, so "**ckp**(P)" succeeds and c_1 is executed. The other is that some permissions in P are disabled, but are not accounted for in Q. So in this case, one cannot determine statically which branch of **ckp** will be taken at runtime. Therefore, R1 conservatively considers typing both branches.

When adapting BN system to Android, R1 is still too strong in some scenarios, especially when it is desired that the *absence* of some permissions leads to the release of sensitive values. Consider for example an application that provides location tracking information related to a certain *advertising ID* (Listing 6.2), where the latter provides a

unique ID for the purpose of anonymizing mobile users to be used for advertising (instead of relying on hardware device IDs such as IMEI numbers). If one can correlate an advertising ID with a unique hardware ID, it will defeat the purpose of the anonymizing service provided by the advertising ID. To prevent that, *getInfo* returns the location information for an advertising ID only if the caller *does not* have access to device ID. To simplify discussion, let us assume that the permissions to access IMEI and location information are denoted by p and q, respectively. Here *id* denotes a unique advertising

```
1 String getInfo() {
2     String r = "";
3     ckp(p) {
4       ckp(q) r = loc; else r = "";
5     } else  {
6       ckp(q) r = id++loc; else r = "";
7     }
8     ret r;
9 }
```

LISTING 6.2: An example about non-monotonic policy.

ID generated and stored by the app for the purpose of anonymizing user tracking and *loc* denotes location information. The function first checks whether the caller has access to IMEI number. If it does, and if it has access to location, then only the location information is returned. If the caller has no access to IMEI number but can access location information, then the combination of advertising id and location *id*++*loc* is returned. In all the other cases, an empty string is returned. Let us consider a lattice with four elements ordered as: $L \leq l_1, l_2 \leq H$, where l_1 and l_2 are incomparable. We specify that empty string is of type L, *loc* is of type l_1, *id* is of type l_2, and the aggregate *id*++*loc* is of type H. Consider the case where the caller has permissions p and q and both are (explicitly) *enabled*. When applying BN system, the desired type of *getInfo* in this case is $() \xrightarrow{\emptyset} l_1$. This means that the type of *r* has to be at most l_1. Since no permissions are disabled, only R1 is applicable to type this program. This, however, will force both branches of **ckp**(p) to have the same type. As a result, *r* has to be typed as H so that all four assignments in the program can be typed.

The issue with the example in Listing 6.2 is that the stated security policy is *non-monotonic* in the sense that an app with more permissions does not necessarily have access to information with a higher level of security. The fact that BN system cannot

precisely capture non-monotonic policies appears to be a design decision: they cited in [148] the lack of motivating examples for non-monotonic policies, and suggested that to accommodate such policies one might need to consider a notion of declassification. As we have seen, however, non-monotonic policies can arise naturally in mobile apps. In a study on Android malwares [12], Enck et. al. identified combinations of permissions that are potentially 'dangerous', in the sense that they allow potentially unsafe information flow. An information flow policy that requires the *absence* of such combinations of permissions in information release would obviously be non-monotonic. In general, non-monotonic policies can be required to solve the *aggregation problem* studied in the information flow theory [149], where several pieces of low security labels information may be pooled together to learn information at higher security labels.

We therefore designed a more precise type system for information flow under an access control model inspired by Android framework. Our type system solves the problem of typing non-monotonic policies without resorting to downgrading or declassifying information. It is done technically via a *merging* operator on security types, to keep information related to both branches of **ckp**. Additionally, there is a significant difference in the permission model used in traditional type systems such as BN system, where permissions are propagated across method invocations among apps. This is due to the fact that permissions in Android are relevant only during inter-process calls, while permissions are not inherited along the call chains across apps. As we shall see in §6.2.5, this may give rise to a type of attack which we call "parameter laundering" attack if one adopts a naive typing rule for function calls. The soundness proof for our type system is significantly different from that for BN type system due to the difference in permission model and the new merging operator on types in our type system.

6.1.2 Contributions

The contributions of this chapter are threefold.

1. We develop a lightweight type system where security types are dependent on a permission-based access control mechanism, and prove its soundness in terms of

non-interference (§6.2). A novel feature of our type system is the type merging constructor, used for typing the conditional branch for permission checking. This makes it possible to model non-monotonic information flow policies.

2. We identify a problem of explicit flow through function calls in the setting where permissions are not propagated during calls. This problem arises as a byproduct of Android's permission model, which is significantly different from that in JVM, and adopting a standard typing rule for function calls such as the one proposed for Java in [148] would result in unsoundness. We name this the parameter laundering problem and propose a typing rule for function calls that prevents it.

3. We provide a type inference for our proposed type system, and show its decidability by reducing it to a constraint solving problem (§6.3).

6.2 A Secure Information Flow Type System

This section presents the proposed information flow type system. §6.2.1 discusses informally a permission-based access control model, which is inspired by the permission mechanism in Android. §6.2.2 and §6.2.3 give the operational semantics of a simple imperative language that includes permission checking constructs based on the abstract permission model. §6.2.4 and §6.2.5 describe our type system and prove its soundness in terms of non-interference.

6.2.1 A Model of Permission-based Access Control

Instead of taking all the language features and the library dependencies of Android apps into account, we focus on the permission model used in inter-component communications within and across apps. Such permissions are used to regulate access to protected resources, such as device id, location information, contact information, etc.

In Android, an app specifies its required permissions at installation time via a manifest file. In recent versions of Android (since Android 6.0, API level 23), some of these permissions need to be granted by users at runtime. But at no point a permission

request is allowed if it is not already specified in the manifest. For now, we assume a permission enforcement mechanism that applies to Android versions prior to version 6.0, so it does not account for permission granting at runtime. To be specific, runtime permission request requires the compatible version specified in the manifest file to be greater than or equal to API level 23, and the running OS should be at least Android 6.0 [150]. Runtime permission granting poses some problems in typing non-monotonic policies, which shall be discussed further in §7.2.

An Android app may provide services to other apps, or other components within the app itself. Such a service provider may impose permissions on other apps who want to access its services. Communication between apps is implemented through Binder IPC (inter-process communications) [151].

In our model, a program is an abstracted version of an app, and the intention is to show how one can reason about information flow in such a service provider when access control is imposed on the calling app. In the following we will not model explicitly the IPC mechanism of Android, but instead model it as a function call. Note that this abstraction is practical since it can be achieved by conventional data and control flow analyses, with the extra modeling of Android IPC specific APIs. The feasibility has been demonstrated by frameworks like FlowDroid [144], Amandroid [145], IccTA [146], etc.

One significant issue that has to be taken into account is that Android framework does not track IPC call chains between apps and permissions of an app are not propagated to the callee. That is, an app A calling another app B does not grant B the permissions which have been assigned to A. This is different from the traditional type systems such as BN where permissions can potentially propagate along the call stacks. Note however that B can potentially have more permissions than A, leading to a potential privilege escalation, a known weaknesses in Android permission system [152]. Another consequence of lacking transitivity is that in designing the type system, one must take care to avoid what we call a "parameter laundering" attack (c.f. §6.2.4).

6.2.2 A Language with Permission Checks

As mentioned earlier, we do not model directly all the language features of an Android app, but use a much simplified language to focus on the permission mechanism part. The language is a variant of the language considered in [153], extended with functions and an operator for permission checks.

We model an *app* as a collection of *functions* (*services*), together with a statically assigned permission set. A *system*, denoted by $\mathcal{S}$, consists of a set of apps. We use capital letters $A, B, \ldots$ to denote apps. A function f defined in an app A is denoted by $A.f$, and may be called independently of other functions in the same app. The intention is that a function models an application component (i.e., *Activity*, *Service*, *BroadCastReceiver*, and *ContentProvider*) in Android, which may be called from within the same app or other apps.

We assume that only one function is executed at a time, so we do not model concurrent executions of apps. We think that in the Android setting, considering sequential behavior only is not overly restrictive. This is because the communication between apps is (mostly) done via IPC. Shared states between apps, which is what contributes to the difficulty in concurrency handling, is mostly absent, apart from the very limited sharing of preferences. In such a setting, each invocation of a service can be treated independently as there is usually no synchronization needed between different invocations. Additionally, we assume functions in a system are not (mutually) recursive, so there is a finite chain of function calls from any given function. The absence of recursion is not a restriction, since our functions are supposed to model communications in Android, which are rarely recursive. We denote with $\mathbf{P}$ the finite set containing all permissions in the system. Each app is assigned a static set of permissions drawn from this set. The powerset of $\mathbf{P}$ is written as $\mathscr{P}$.

For simplicity, we consider only programs manipulating *integers*, so the expressions in our language all have the integer type. Boolean values are encoded as 0 (false) and any non-zero values (true). The grammar for expressions is given below:

$$e ::= n \mid x \mid e \circ e$$

where n denotes an integer literal, x denotes a variable, and $\circ$ denotes a binary operation. The commands of the language are given in the following grammar:

$$\begin{aligned} c \ ::= \ & x := e \mid \textbf{if } e \textbf{ then } c \textbf{ else } c \mid \textbf{while } e \textbf{ do } c \mid c; c \\ & \mid \textbf{letv } x = e \textbf{ in } c \mid x := \textbf{Icall } A.f(\overline{e}) \mid \textbf{ckp}(p)\ c \textbf{ else } c \end{aligned}$$

The first four constructs are assignment, conditional, while-loop, and sequential composition, respectively. The statement "**letv** $x = e$ **in** c" is a local variable declaration statement. Here x is declared and initialized to e, and its scope is the command c. We require that x does not occur in e. The statement "$x :=$ **Icall** $A.f(\overline{e})$" denotes an assignment whose right hand side is a function call to $A.f$. The statement "**ckp**(p) c_1 **else** c_2" checks whether the calling app has permission p: if so then c_1 is executed, otherwise c_2 is executed. This is similar to the **test** primitive in BN system, except that we allow checking only one permission at a time. This is a not real restriction since both primitives can simulate one another.

A function declaration has the following syntax:

$$F ::= A.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{ret } r\}\}$$

where $A.f$ is the function name, $\bar{x}$ are function parameters, c is a command, and r is a local variable that holds the return value of the function. $\overline{x}$ and r are bound variables with the command "$c;$ **ret** r" in their scopes. We consider only *closed functions*, i.e., the variables occurring in c are either introduced by **letv** or from the set $\{\overline{x}, r\}$.

6.2.3 Operational Semantics

We assume that function definitions are stored in a table $\mathcal{F}d$ indexed by function names, and the permission sets assigned to apps are given by a table Θ indexed by app names.

An *evaluation environment* is a finite mapping from variables to values (i.e., integers in the simplified language). We denote with $EEnv$ the set of evaluation environments. Elements of $EEnv$ are ranged over by μ. We use the notation $[x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$ to denote an evaluation environment mapping variable x_i to value v_i; this will sometimes be abbreviated as $[\overline{x} \mapsto \overline{v}]$. The domain of $\mu = [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]$

$$\text{E-VAL}\ \frac{}{\mu \vdash v \rightsquigarrow v} \qquad \text{E-VAR}\ \frac{}{\mu \vdash x \rightsquigarrow \mu(x)}$$

$$\text{E-OP}\ \frac{\mu \vdash e_1 \rightsquigarrow v_1 \quad \mu \vdash e_2 \rightsquigarrow v_2}{\mu \vdash e_1 \circ e_2 \rightsquigarrow v_1 \circ v_2} \qquad \text{E-LETV}\ \frac{\mu \vdash e \rightsquigarrow v \quad \mu[x \mapsto v]; A; P \vdash c \rightsquigarrow \mu'}{\mu; A; P \vdash \textbf{letv}\ x = e\ \textbf{in}\ c \rightsquigarrow \mu' - x}$$

$$\text{E-SEQ}\ \frac{\mu; A; P \vdash c_1 \rightsquigarrow \mu' \quad \mu'; A; P \vdash c_2 \rightsquigarrow \mu''}{\mu; A; P \vdash c_1; c_2 \rightsquigarrow \mu''} \qquad \text{E-ASS}\ \frac{\mu \vdash e \rightsquigarrow v}{\mu; A; P \vdash x := e \rightsquigarrow \mu[x \mapsto v]}$$

$$\text{E-IF-T}\ \frac{\mu \vdash e \rightsquigarrow v \quad v \neq 0 \quad \mu; A; P \vdash c_1 \rightsquigarrow \mu'}{\mu; A; P \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow \mu'} \qquad \text{E-IF-F}\ \frac{\mu \vdash e \rightsquigarrow v \quad v = 0 \quad \mu; A; P \vdash c_2 \rightsquigarrow \mu'}{\mu; A; P \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow \mu'}$$

$$\text{E-WHILE-T}\ \frac{\mu \vdash e \rightsquigarrow v \quad v \neq 0 \quad \mu; A; P \vdash c \rightsquigarrow \mu' \quad \mu'; A; P \vdash \textbf{while}\ e\ \textbf{do}\ c \rightsquigarrow \mu''}{\mu; A; P \vdash \textbf{while}\ e\ \textbf{do}\ c \rightsquigarrow \mu''} \qquad \text{E-WHILE-F}\ \frac{\mu \vdash e \rightsquigarrow v \quad v = 0}{\mu; A; P \vdash \textbf{while}\ e\ \textbf{do}\ c \rightsquigarrow \mu}$$

$$\text{E-CP-T}\ \frac{p \in P \quad \mu; A; P \vdash c_1 \rightsquigarrow \mu'}{\mu; A; P \vdash \textbf{ckp}(p)\ c_1\ \textbf{else}\ c_2 \rightsquigarrow \mu'} \qquad \text{E-CP-F}\ \frac{p \notin P \quad \mu; A; P \vdash c_2 \rightsquigarrow \mu'}{\mu; A; P \vdash \textbf{ckp}(p)\ c_1\ \textbf{else}\ c_2 \rightsquigarrow \mu'}$$

$$\text{E-ICALL}\ \frac{\mathcal{F}d(B.f) = B.f(\overline{y})\{\textbf{init}\ r = 0\ \textbf{in}\ \{c; \textbf{ret}\ r\}\} \qquad \mu \vdash \overline{e} \rightsquigarrow \overline{v} \quad [\overline{y} \mapsto \overline{v}, r \mapsto 0]; B; \Theta(A) \vdash c \rightsquigarrow \mu'}{\mu; A; P \vdash x := \textbf{Icall}\ B.f(\overline{e}) \rightsquigarrow \mu[x \mapsto \mu'(r)]}$$

FIGURE 6.1: Evaluation rules for expressions and commands, in the presence of function definition table $\mathcal{F}d$ and permission assignment Θ.

(i.e., $\{x_1, \ldots, x_n\}$) is denoted by $dom(\mu)$. Given two environments μ_1 and μ_2, we define $\mu_1\mu_2$ as the environment μ such that $\mu(x) = \mu_2(x)$ if $x \in dom(\mu_2)$, otherwise $\mu(x) = \mu_1(x)$. For example, $\mu[x \mapsto v]$ maps x to v, and y to $\mu(y)$ for any $y \in dom(\mu)$ such that $y \neq x$. Given a mapping μ and a variable x, we write $\mu - x$ to denote the mapping resulting from removing x from $dom(\mu)$.

The operational semantics for expressions and commands is given in Figure 6.1. The evaluation judgment for expressions has the form $\mu \vdash e \rightsquigarrow v$, which states that expression e evaluates to value v when variables in e are interpreted in the evaluation environment μ. We write $\mu \vdash \overline{e} \rightsquigarrow \overline{v}$, where $\overline{e} = e_1, \ldots, e_n$ and $\overline{v} = v_1, \ldots, v_n$ for some n, to denote a sequence of judgments $\mu \vdash e_1 \rightsquigarrow v_1, \ldots, \mu \vdash e_n \rightsquigarrow v_n$.

The evaluation judgment for commands takes the form $\mu; A; P \vdash c \rightsquigarrow \mu'$ where μ is an evaluation environment before the execution of the command c, and μ' is the evaluation environment after the execution of c. Here A refers to the app to which the command c belongs. The permission set P denotes the *permission context*, i.e., it is the set of permissions of the app which invokes the function of A in which the command c resides. The caller app may be A itself (in which case the permission context will be the same as the permission set of A) but more often it is another app in the system.

The operational semantics of most commands are straightforward. We explain the semantics of the *ckp* primitive and the function call. Rules (E-CP-T) and (E-CP-F) capture the semantics of the **ckp** primitive. These are where the permission context P in the evaluation judgement is used. The semantics of function calls is given by (E-ICALL). Notice that c inside the body of *callee* is executed under the permission context $\Theta(A)$, which is the permission set of A. The permission context P in the conclusion of that rule, which denotes the permission of the app that calls A, is not used in the premise. That is, the permission context of A is not inherited by the callee function $B.f$. This reflects the way permission contexts in Android are passed on during IPCs [151,154], and is also a major difference between our permission model and that in BN type system, where permission contexts are inherited by successive function calls.

6.2.4 Security Types

In information flow type systems such as [153], it is common to adopt a lattice structure to encode security labels. Security types in this setting are just security levels. In our case, we generalize the security types to account for the dependency of security labels on permissions. So we shall distinguish security labels, given by a lattice structure which encodes sensitivity levels of information, and security types, which are mappings from permissions to security labels. We assume the security labels are given by a lattice $\mathscr{L}$, with a partial order $\leq_{\mathscr{L}}$. Security types are defined in the following.

Definition 6.1. A *base security type* (or *base type*) t is a mapping from $\mathscr{P}$ to $\mathscr{L}$. We denote with $\mathcal{T}$ the set of base types. Given two base types s and t, we say $s = t$ iff $s(P) = t(P)$ for all $P \in \mathscr{P}$. We define an ordering $\leq_{\mathcal{T}}$ on base types as follows: $s \leq_{\mathcal{T}} t$ iff $\forall\ P \in \mathscr{P}$, $s(P) \leq_{\mathscr{L}} t(P)$.

Lemma 6.2. *$\leq_{\mathcal{T}}$ is a partial order relation on $\mathcal{T}$.*

Proof. **Reflectivity** $\forall P, t(P) \leq_{\mathscr{L}} t(P)$ therefore $t \leq_{\mathcal{T}} t$.

Antisymmetry $t \leq_{\mathcal{T}} s \wedge s \leq_{\mathcal{T}} t \iff \forall P, t(P) \leq_{\mathscr{L}} s(P) \wedge s(P) \leq_{\mathscr{L}} t(P)$ therefore $\forall P, t(P) = s(P)$, which means that $s = t$.

Transitivity if $r \leq_{\mathcal{T}} s$ and $s \leq_{\mathcal{T}} t$, $\forall P, r(P) \leq_{\mathscr{L}} s(P)$ and $s(P) \leq_{\mathscr{L}} t(P)$ therefore $r(P) \leq_{\mathscr{L}} t(P)$, which means that $r \leq_{\mathcal{T}} t$.

□

As we shall see, if a variable is typed by a base type, the sensitivity of its content may depend on the app's permissions which writes to the variable. In contrast, in traditional information flow type systems, a variable annotated with a security label has a fixed sensitivity level regardless of the permissions of the app that writes to the variable.

The set of base types with the order $\leq_{\mathcal{T}}$ forms a lattice. The join and meet of the lattice are defined as follows:

Definition 6.3. For $s, t \in \mathcal{T}$, $s \sqcup t$ and $s \sqcap t$ are defined as

$$(s \sqcup t)(P) = s(P) \sqcup t(P), \forall P \in \mathscr{P}$$
$$(s \sqcap t)(P) = s(P) \sqcap t(P), \forall P \in \mathscr{P}$$

Lemma 6.4. *Given two base types s and t, it follows that*

(a) $s \leq_{\mathcal{T}} s \sqcup t$ *and* $t \leq_{\mathcal{T}} s \sqcup t$.

(b) $s \sqcap t \leq_{\mathcal{T}} s$ *and* $s \sqcap t \leq_{\mathcal{T}} t$.

Proof. Immediately from Definition 6.1. □

Lemma 6.5. $(\mathcal{T}, \leq_{\mathcal{T}})$ *forms a lattice.*

Proof. $\forall s, t \in \mathscr{P}$, according to Lemma 6.4, $s \sqcup t$ is their upper bound. Suppose r is another upper bound, i.e., $s \leq_{\mathcal{T}} r$ and $t \leq_{\mathcal{T}} r$, which means $\forall P \in \mathscr{P}, (s \sqcup t)(P) = s(P) \sqcup t(P) \leq_{\mathscr{L}} r(P)$, so $s \sqcup t \leq r$. Therefore $s \sqcup t$ is the least upper bound of $\{s, t\}$. Similarly, $s \sqcap t$ is s and t's greatest lower bound. This makes $(\mathcal{T}, \leq_{\mathcal{T}})$ a lattice. □

From now on, we shall drop the subscripts in $\leq_{\mathscr{L}}$ and $\leq_{\mathcal{T}}$ when no ambiguity arises.

Definition 6.6. Given a security label l, we define $\hat{l}$ as: for all $P \in \mathscr{P}$, we have $\hat{l}(P) = l$.

Accordingly, a security label l can be lifted to the base type $\hat{l}$ that maps all permission sets to level l itself.

Definition 6.7. A *function type* has the form $\overline{t} \to t$, where $\overline{t} = (t_1, \ldots, t_m)$, $m \geq 0$ and t, t_i are base types. The types $\overline{t}$ are the types for the arguments of the function and t is the return type of the function.

Lemma 6.8. *Given $P \in \mathscr{P}$ and $p \in \mathbf{P}$,*

(a) If $p \in P$, then $(t \uparrow_p)(P) = t(P)$.

(b) If $p \notin P$, then $(t \downarrow_p)(P) = t(P)$.

Proof. If $p \in P$, $P \cup \{p\} = P$, therefore $(t \uparrow_p)(P) = t(P \cup \{p\}) = t(P)$; if $p \notin P$, $P \setminus \{p\} = P$, therefore $(t \downarrow_p)(P) = t(P \setminus \{p\}) = t(P)$. □

Lemma 6.9. *If $s \leq t$, then $s \uparrow_p \leq t \uparrow_p$ and $s \downarrow_p \leq t \downarrow_p$.*

Proof. For $P \in \mathbf{P}$, since $s \leq t$, $s(P \cup \{p\}) \leq t(P \cup \{p\})$ and $s(P \setminus \{p\}) \leq t(P \setminus \{p\})$, according to Definition 6.1 and Definition 6.10, the conclusion follows. □

In our type system, security types of expressions (commands, functions, resp.) may be altered depending on the execution context. That is, when an expression is used in a context where a permission check has been performed (either successfully or unsuccessfully), its type may be adjusted to take into account the *presence* or *absence* of the checked permission. Such an adjustment is called a *promotion* or a *demotion*.

Definition 6.10. Given a permission p, the *promotion* and *demotion* of a base type t with respect to p are:

$$(t \uparrow_p)(P) = t(P \cup \{p\}), \forall P \in \mathscr{P} \quad \text{(promotion)}$$

$$(t \downarrow_p)(P) = t(P \setminus \{p\}), \forall P \in \mathscr{P} \quad \text{(demotion)}$$

The *promotion* and *demotion* of $\overline{t} \to t$, where $\overline{t} = (t_1, \ldots, t_m)$, are respectively:

$$(\overline{t} \to t) \uparrow_p = \overline{t} \uparrow_p \to t \uparrow_p, \text{ where } \overline{t} \uparrow_p = (t_1 \uparrow_p, \ldots, t_m \uparrow_p),$$

$$(\overline{t} \to t) \downarrow_p = \overline{t} \downarrow_p \to t \downarrow_p, \text{ where } \overline{t} \downarrow_p = (t_1 \downarrow_p, \ldots, t_m \downarrow_p).$$

$$\text{T-VAR}\frac{}{\Psi \vdash x : \Psi(x)} \qquad \text{T-OP}\frac{\Psi \vdash e_1 : t \quad \Psi; A \vdash e_2 : t}{\Psi \vdash e_1 \circ e_2 : t}$$

$$\text{T-SUB}_e\frac{\Psi \vdash e : s \quad s \leq t}{\Psi \vdash e : t} \qquad \text{T-SUB}_c\frac{\Psi; A \vdash c : s \quad t \leq s}{\Psi; A \vdash c : t}$$

$$\text{T-ASS}\frac{\Psi \vdash e : \Psi(x)}{\Psi; A \vdash x := e : \Psi(x)} \qquad \text{T-LETV}\frac{\Psi \vdash e : s \quad \Psi[x : s]; A \vdash c : t}{\Psi; A \vdash \textbf{letv}\ x = e\ \textbf{in}\ c : t}$$

$$\text{T-IF}\frac{\Psi \vdash e : t \quad \Psi; A \vdash c_1 : t \quad \Psi; A \vdash c_2 : t}{\Psi; A \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : t} \qquad \text{T-CP}\frac{\Psi \uparrow_p; A \vdash c_1 : t_1 \quad \Psi \downarrow_p; A \vdash c_2 : t_2}{\Psi; A \vdash \textbf{ckp}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \triangleright_p t_2}$$

$$\text{T-WHILE}\frac{\Psi \vdash e : t \quad \Psi; A \vdash c : t}{\Psi; A \vdash \textbf{while}\ e\ \textbf{do}\ c : t} \qquad \text{T-SEQ}\frac{\Psi; A \vdash c_1 : t \quad \Psi; A \vdash c_2 : t}{\Psi; A \vdash c_1; c_2 : t}$$

$$\text{T-ICALL}\frac{\mathcal{F}t(B.f) = \overline{t} \to t' \quad \Psi \vdash \overline{e} : \pi_{\Theta(A)}(\overline{t}) \quad \pi_{\Theta(A)}(t') \leq \Psi(x)}{\Psi; A \vdash x := \textbf{Icall}\ B.f(\overline{e}) : \Psi(x)}$$

$$\text{T-FUN}\frac{[\overline{x} : \overline{t}, r : t']; B \vdash c : s}{\vdash B.f(\overline{x})\{\textbf{init}\ r = 0\ \textbf{in}\ \{c; \textbf{ret}\ r\}\} : \overline{t} \to t'}$$

FIGURE 6.2: Typing rules for expressions, commands, and functions.

6.2.5 Security Type System

We first define operations on security types and permissions which will be used later.

Definition 6.11. Given $t \in \mathcal{T}$ and $P \in \mathscr{P}$, the *projection* of t on a permission set P is a security type $\pi_P(t)$ defined as:

$$\pi_P(t)(Q) = t(P),\ \forall Q \in \mathscr{P}.$$

Type projection of a list of types on P is then written as

$$\pi_P((t_1, \ldots, t_n)) = (\pi_P(t_1), \ldots, \pi_P(t_n)).$$

Definition 6.12. Given a permission p and two types t_1 and t_2, the *merging* of t_1 and t_2 along p, denoted as $t_1 \triangleright_p t_2$, is:

$$(t_1 \triangleright_p t_2)(P) = \begin{cases} t_1(P) & p \in P \\ t_2(P) & p \notin P \end{cases} \quad \forall P \in \mathscr{P}$$

A *typing environment* is a finite mapping from variables to base types. We use the notation $[x_1 : t_1, \ldots, x_n : t_n]$ to enumerate a typing environment with domain

$\{x_1, \ldots, x_n\}$. Typing environments are ranged over by Ψ. Given Ψ_1 and Ψ_2 such that $dom(\Psi_1) \cap dom(\Psi_2) = \emptyset$, we write $\Psi_1\Psi_2$ to denote a typing environment that is the (disjoint) union of the mappings in Ψ_1 and Ψ_2.

Definition 6.13. Given a typing environment Ψ, its *promotion* and *demotion* along p are typing environments $\Psi\uparrow_p$ and $\Psi\downarrow_p$, such that $(\Psi\uparrow_p)(x) = \Psi(x)\uparrow_p$ and $(\Psi\downarrow_p)(x) = \Psi(x)\downarrow_p$ for every $x \in dom(\Psi)$. The projection of Ψ on $P \in \mathscr{P}$ is a typing environment $\pi_P(\Psi)$ such that $(\pi_P(\Psi))(x) = \pi_P(\Psi(x))$ for each $x \in dom(\Psi)$.

There are three typing judgments in our type system as explained below. All these judgments are implicitly parameterized by a function type table, $\mathcal{F}t$, which maps all function names to function types, and a mapping Θ assigning permission sets to apps.

- Expression typing: $\Psi \vdash e : t$. This says that under Ψ, the expression e has a base type at most t.
- Command typing: $\Psi; A \vdash c : t$. It means that the command c writes to variables with type at least t, when executed by app A, under the typing environment Ψ.
- Function typing: The typing judgment takes the form:

$$\vdash B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{ret } r\}\} : \overline{t} \to t'$$

  where $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{t} = (t_1, \ldots, t_n)$ for some $n \geq 0$. Functions are polymorphic in the permissions of the caller. Intuitively, each caller of the function above with permission set P "sees" the function as having type $\pi_P(\overline{t}) \to \pi_P(t')$. That is, if the function is called from another app with permission P, then it expects input of type up to $\pi_P(\overline{t})$ and a return value of type at most $\pi_P(t')$.

The typing rules are given in Figure 6.2. Most of them are common to information flow type systems [148, 153, 155] except for T-CP and T-ICALL. Note that in the subtyping rule for commands (T-SUB$_c$), the security type of the effect of the command can be safely downgraded, since typing for commands keeps track of a lower bound of the write effects of the command. This typing rule for command is standard, see, e.g., [153] for a more detailed discussion.

In T-CP, to type **ckp**(p) c_1 **else** c_2, we type c_1 in a promoted typing environment for a successful permission check on p, and c_2 in a demoted typing environment for a failed permission check on p. The challenge is how to combine the types of the two premises to obtain the type for the conclusion. One possibility is to force the type of the two premises and the conclusion to be identical (i.e., treat permission check the same as other if-then-else statements and apply T-IF). This, as we have seen in §6.1, leads to a loss in precision of the type for **ckp** construct. We consider a more refined *merged* type $t_1 \triangleright_p t_2$ for the conclusion, where t_1 (t_2 resp.) is the type of the left (right resp.) premise. To understand the merged type, consider a scenario where the statement is executed in a context where permission p is *present*. The permission check succeeds and **ckp**(p) c_1 **else** c_2 is equivalent to c_1. In this case, one would expect that the behavior of **ckp**(p) c_1 **else** c_2 would be equivalent to that of c_1. This is in fact captured by the equation $(t_1 \triangleright_p t_2)(P) = t_1(P)$ for all P such that $p \in P$, which holds by definition. A dual scenario arises when p is not in the permissions of the execution context.

In T-ICALL, the callee function $B.f$ is assumed to be type checked beforehand and its type is given in the FT table. Here the function $B.f$ is called by A so the type of $B.f$ as seen by A should be a projection of the type given in $FT(B.f)$ on the permissions of A (given by $\Theta(A)$): $\pi_{\Theta(A)}(\overline{t}) \rightarrow \pi_{\Theta(A)}(t')$. Therefore the arguments for the function call should be typed as $\Psi \vdash \overline{e} : \pi_{\Theta(A)}(\overline{t})$ and the return type (as viewed by A) should be dominated by the type of x, i.e., $\pi_{\Theta(A)}(t') \leq \Psi(x)$.

Parameter laundering It is essential that in Rule T-ICALL, the arguments $\overline{e}$ and the return value of the function call are typed according to the projection of $\overline{t}$ and t' on $\Theta(A)$. If they are instead typed with $\overline{t}$, then there is a potential implicit flow via a "parameter laundering" attack. To see why, consider the following alternative to T-ICALL:

$$\text{T-ICALL'}\ \frac{\mathcal{F}t(B.f) = \overline{t} \rightarrow t' \qquad \Psi \vdash \overline{e} : \overline{t} \qquad t' \leq \Psi(x)}{\Psi; A \vdash x := \textbf{Icall}\ B.f(\overline{e}) : \Psi(x)}$$

Notice that the type of the argument $\overline{e}$ must match the type of the formal parameter of the function $B.f$. This is essentially adopted by BN system for method calls [148].

Let us consider the example in Listing 6.3. Let $\mathbf{P} = \{p\}$ and t be the base type $t = \{\emptyset \mapsto L, \{p\} \mapsto H\}$, where L and H are bottom and top levels respectively. Here we assume *P_INFO* is a sensitive value of security labels H that needs to be protected,

```
1  A.f(x) {   // A does not have permission p
2    init r = 0 in { r := call B.g (x); ret r }
3  }
4
5  B.g(x) { // B does not have permission p
6    init r = 0 in {
7      ckp(p) r := 0 else r := x;
8      ret r
9    }
10 }
11
12 C.getsecret() { // C has permission p
13   init r = 0 in {
14     ckp(p) r := P_INFO else r := 0;
15     ret r
16   }
17 }
18
19 M.main() { // M has permission p
20     init r = 0 in {
21     letv x_H = 0 in
22       {x_H := C.getsecret(); r := call A.f(x_H)};
23       ret r
24   }
25 }
```

LISTING 6.3: An example illustrating the parameter laundering issue.

so function *C.getsecret* is required to have type $() \to t$. That is, only apps that have the required permission p may obtain the secret value. Suppose the permissions assigned to the apps are given by: $\Theta(A) = \Theta(B) = \emptyset, \Theta(C) = \Theta(M) = \{p\}$.

If we were to adopt the modified T-ICALL' instead of T-ICALL, then we can assign the following types to the above functions:

$$FT := \begin{cases} A.f & \mapsto \quad t \to \hat{L} \\ B.g & \mapsto \quad t \to \hat{L} \\ C.getsecret & \mapsto \quad () \to t \\ M.main & \mapsto \quad () \to \hat{L} \end{cases}$$

Notice that the return type of $M.main$ is $\hat{L}$ despite having a return value that contains sensitive value *P_INFO*. If we were to use T-ICALL' in place of T-ICALL, the above functions can be typed as shown in Figure 6.3. Finally, still assuming T-ICALL', a partial typing derivation for $M.main$ is given in Figure 6.4.

As shown in Figure 6.3, *B.g* can be given type $t \to \hat{L}$. Intuitively, it checks that

the caller has permission p. If it does, then *B.g* returns 0 (non-sensitive), otherwise it returns the argument of the function (i.e., x). This is as expected and is sound, under the assumption that the security labels of the content of x is dependent on the permissions of the caller. If the caller of *B.g* is the original creator of the content of x, then the assumption is trivially satisfied. The situation gets a bit tricky when the caller simply passes on the content it receives from another app to x. In our example, app A makes a call to *B.g*, and passes on the value of x it receives. In the run where *A.f* is called from *M.main*, the value of x is actually *sensitive* since it requires the permission p to acquire. However, when it goes through *A.f* to *B.g*, the value of x is perceived as *non-sensitive* by B, since the caller in this case (A) has no permissions. The use of the intermediary A in this case in effect launders the permissions associated with x. Therefore, if the rule T-ICALL' is used in place of T-ICALL, the call chain from *M.main* to *A.f* and finally to *B.g* can all be typed. This is correct in a setting where permissions are *propagated* along with calling context (e.g., [148]) however it is incorrect in the Android permission model (§6.2.1). To avoid the parameter laundering problem, our approach ensures that an app may only pass an argument to another function if the app itself is authorized to access the content of the argument in the first place, as formalized in the rule T-ICALL.

$$\text{T-FUN}\ \frac{\text{T-ICALL'}\ \dfrac{FT(B.g) = t \to \hat{L} \qquad x : t, r : \hat{L} \vdash x : t \qquad \hat{L} \leq t}{x : t, r : \hat{L}; A \vdash r := \textbf{Icall}\ B.g(x) : \hat{L}}}{\vdash A.f(x)\{\textbf{init}\ r = 0\ \textbf{in}\ \{r := \textbf{Icall}\ B.g(x); \textbf{ret}\ r\}\} : t \to \hat{L}}$$

$$\text{T-FUN}\ \frac{\text{T-CP}\ \dfrac{x : t \uparrow_p, r : \hat{L} \uparrow_p \vdash r := 0 : \hat{L} \qquad x : t \downarrow_p, r : \hat{L} \downarrow_p \vdash r := x : \hat{L}}{x : t, r : \hat{L} \vdash \textbf{ckp}(p)\ r := 0\ \textbf{else}\ r := x : \hat{L} \triangleright_p \hat{L}}}{\vdash B.g(x)\{\textbf{init}\ r = 0\ \textbf{in}\ \{\textbf{ckp}(p)\ r := 0\ \textbf{else}\ r := x; \textbf{ret}\ r\}\} : t \to \hat{L}}$$

Note that $t \uparrow_p = \hat{H}$, $t \downarrow_p = \hat{L} = \hat{L} \downarrow = \hat{L} \uparrow$ and $\hat{L} \triangleright_p \hat{L} = \hat{L}$.

$$\text{T-FUN}\ \frac{\text{T-CP}\ \dfrac{r : t \uparrow_p \vdash r := \text{SECRET} : \hat{H} \qquad r : t \downarrow_p \vdash r := 0 : \hat{L}}{r : t \vdash \textbf{ckp}(p)\ r := \text{SECRET}\ \textbf{else}\ r := 0 : \hat{H} \triangleright_p \hat{L}}}{\vdash C.getsecret(\,)\{\textbf{init}\ r = 0\ \textbf{in}\ \{\textbf{ckp}(p)\ r := \text{SECRET}\ \textbf{else}\ r := 0; \textbf{ret}\ r\}\} : () \to t}$$

Note that $\hat{H} \triangleright_p \hat{L} = t$.

FIGURE 6.3: Typing derivations for A.f, B.g and C.getsecret

$$\text{T-LETV}\ \frac{r : \hat{L} \vdash 0 : t \qquad \text{T-SEQ}\ \dfrac{\Psi; M \vdash x_H := \textbf{Icall}\ C.getsecret(\,) : \hat{L} \qquad \Psi; M \vdash r := \textbf{Icall}\ A.f(x_H) : \hat{L}}{r : \hat{L}, x_H : t; M \vdash x_H := \textbf{Icall}\ C.getsecret(\,); r := \textbf{Icall}\ A.f(x_H) : \hat{L}}}{\text{T-FUN}\ \dfrac{r : \hat{L}\ ; M \vdash \textbf{letv}\ x_H = 0\ \textbf{in}\ x_H := \textbf{Icall}\ C.getsecret(\,); r := \textbf{Icall}\ A.f(x_H) : \hat{L}}{\begin{array}{l}\vdash M.main(\,)\ \{\textbf{init}\ r = 0\ \textbf{in} \\ \quad \textbf{letv}\ x_H = 0\ \textbf{in}\ \{ \\ \quad\quad x_H := \textbf{Icall}\ C.getsecret(\,); \\ \quad\quad r := \textbf{Icall}\ A.f(x_H) \\ \quad \} \\ \quad \textbf{ret}\ r \quad \}\end{array} : () \to \hat{L}}}$$

where $\Psi = \{r : \hat{L}, x_H : t\}$ and the second and the third leaves are derived, respectively, as follows:

$$\text{T-ICALL'}\ \frac{FT(C.getsecret) = () \to t \qquad \Psi \vdash () : () \qquad t \leq \Psi(x_H) = t}{\text{T-SUB}_c\ \dfrac{\Psi; M \vdash x_H := \textbf{Icall}\ C.getsecret(\,) : t}{\Psi; M \vdash x_H := \textbf{Icall}\ C.getsecret(\,) : \hat{L}}}$$

$$\text{T-ICALL'}\ \frac{FT(A.f) = t \to \hat{L} \qquad \Psi \vdash x_H : t \qquad \hat{L} \leq \Psi(x_H) = t}{\Psi; M \vdash r := \textbf{Icall}\ A.f(x_H) : \hat{L}}$$

FIGURE 6.4: A typing derivation for M.main

With the correct typing rule for function calls, the function $A.f$ cannot be assigned type $t \to \hat{L}$, since that would require the instance of T-ICALL (i.e., when making the call to $B.g$) in this case to satisfy the constraint:

$$x : t, r : \hat{L} \vdash x : \pi_{\Theta(A)}(t)$$

where $\pi_{\Theta(A)}(t) = \hat{L}$, which is impossible since $t \not\leq \hat{L}$. What this means is essentially that in our type system, information received by an app A from the parameters cannot be propagated by A to another app B, unless A is already authorized to access the information contained in the parameter. Note that this only restricts the propagation of such parameters to other apps; the app A can process the information internally without necessarily violating the typing constraints.

Finally, the reader may check that if we fix the type of $B.g$ to $t \to \hat{L}$ then $A.f$ can only be assigned type $\hat{L} \to \hat{L}$. In no circumstances can $M.main$ be typed, since the statement $x_H := C.getsecret()$ forces x_H to have type $\hat{H}$, and thus cannot be passed to $A.f$ as an argument.

6.2.6 Non-interference and Soundness

We first define an *indistinguishability* relation between evaluation environments. Such a definition typically assumes an observer who may observe values of variables at a certain security labels. In the non-dependent setting, the security labels of the observer is fixed, say at l_O, and valuations of variables at level l_O or below are required to be identical. In our setting, the security labels of a variable in a function can vary depending on the permissions of the caller app (which may be the observer itself), so it may seem more natural to define indistinguishability in terms of the permission set assigned to the observer. However, we argue that such a definition is subsumed by the more traditional definition that is based on the security labels of the observer. Assuming that the observer app is assigned a permission set P, then given two variables $x : t$ and $y : t'$, the level of information that the observer can access through x and y is at most $t(P) \sqcap t'(P)$. In general the least upper bound of the security labels that an observer with permission P has access to can be computed from the least upper bound of projections (along P) of the types of variables and the return types of functions in the system. In the following definition of indistinguishability, we simply assume that such an upper bound has been computed, and we will not refer explicitly to the permission set of the observer from which this upper bound is derived.

Definition 6.14. Given evaluation environments μ, μ', a typing environment Ψ, a security label $l_O \in \mathscr{L}$ of the observer, the *indistinguishability relation* $=^{l_O}_{\Psi}$ is defined as:

$$\mu =^{l_O}_{\Psi} \mu' \text{ iff} \forall x \in dom(\Psi).\ \big(\Psi(x) \leq \hat{l_O} \Rightarrow \mu(x) = \mu'(x)\big)$$

where $\mu(x) = \mu'(x)$ holds iff both sides of the equation are defined and equal, or both sides are undefined.

Note that in Definition 6.14, μ and μ' may not have the same domain, but they must agree on their valuations for the variables in the domain of Ψ. Note also that since base types are functions from permissions to security labels, the security level l_O needs to be lifted to a base type in the comparison $\Psi(x) \leq \hat{l_O}$. The latter implies that $\Psi(x)(P) \leq l_O$ (in the latice $\mathscr{L}$) for every permission set P. If the base type of each variable assigns the same security labels to every permission set (i.e., the security labels is independent

of the permissions), then our notion of indistinguishability coincides with the standard definition for the non-dependent setting.

Lemma 6.15. *$=_\Psi^{l_O}$ is an equivalence relation on $EEnv$.*

Proof. **Reflexivity** Obviously $\mu =_\Psi^{l_O} \mu$.

Symmetry Since $\forall x \in dom(\Psi).(\Psi(x) \leq \hat{l}_O \Rightarrow \mu' =_\Psi^{l_O} \mu)$.

Transitivity If $\mu_1 =_\Psi^{l_O} \mu_2$ and $\mu_2 =_\Psi^{l_O} \mu_3$, for a given $x \in dom(\Psi)$, when $\Psi(x) \leq \hat{l_O}$, we have $\mu_1(x) = \mu_2(x)$ and $\mu_2(x) = \mu_3(x)$.

(1) If $\mu_1(x) \neq \bot$, then $\mu_2(x) \neq \bot$ by the first equation, which in return requires $\mu_3(x) \neq \bot$ by the second equation; by transitivity $\mu_1(x) = \mu_3(x)(\neq \bot)$.

(2) If $\mu_1(x) = \bot$, the first equation requires that $\mu_2(x) = \bot$, which makes $\mu_3(x) = \bot$, therefore both $\mu_1(x) = \mu_3(x)(= \bot)$.

Therefore $\mu_1(x) = \mu_3(x)$.

□

Lemma 6.16. *If $\mu =_\Psi^{l_O} \mu'$ then for each $P \in \mathscr{P}$, $\mu =_{\pi_P(\Psi)}^{l_O} \mu'$.*

Proof. $\forall x \in dom(\Psi)$, we need to prove that when $\pi_P(\Psi)(x) \leq l_O$ then $\mu(x) = \mu'(x)$. But $\pi_P(\Psi)(x) = \pi_P(\Psi(x)) = t(P)$, from the definition of $\mu(x) = \mu'(x)$, the conclusion holds. □

We hereby give the definitions for well-typed property (Definition 6.17) and non-interference for the type system (Defintion 6.22 and Definition 6.24), together with the final soundness conclusion (Theorem 6.25).

Definition 6.17. Let $\mathcal{S}$ be a system, and let $\mathcal{F}d$, $\mathcal{F}t$ and Θ be its function declaration table, function type table, and permission assignments. We say $\mathcal{S}$ is *well-typed* iff for every function $A.f, \vdash \mathcal{F}d(A.f) : \mathcal{F}t(A.f)$ is derivable.

Recall that we assume no (mutual) recursions, so every function call chain in a well-typed system is finite; this is formalized via the rank function below. We will use this

as a measure in our soundness proof (Lemma 6.23).

$$
\begin{array}{ll}
\textbf{rank}(x := e) & = 0 \\
\textbf{rank}(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) & = max(\textbf{rank}(c_1), \textbf{rank}(c_2)) \\
\textbf{rank}(c_1; c_2) & = max(\textbf{rank}(c_1), \textbf{rank}(c_2)) \\
\textbf{rank}(\textbf{while } e \textbf{ do } c) & = \textbf{rank}(c) \\
\textbf{rank}(\textbf{letv } x = e \textbf{ in } c) & = \textbf{rank}(c) \\
\textbf{rank}(x := \textbf{Icall } A.f(\overline{e})) & = \textbf{rank}(\mathcal{F}d(A.f)) + 1 \\
\textbf{rank}(\textbf{ckp}(p)\ c_1 \textbf{ else } c_2) & = max(\textbf{rank}(c_1), \textbf{rank}(c_2)) \\
\textbf{rank}(A.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{ret } r\}\}) & = \textbf{rank}(c)
\end{array}
$$

The next two lemmas relate projection, promotion/demotion and indistinguishability relation.

Lemma 6.18. *If $p \in P$, then $\mu =^{l_O}_{\pi_P(\Psi)} \mu'$ iff $\mu =^{l_O}_{\pi_P(\Psi\uparrow_p)} \mu'$.*

Proof. We first note that $dom(\pi_P(\Psi)) = dom(\pi_P(\Psi \uparrow_p))$ since both promotion and projection do not change the domain of a typing environment. We then show below that $\pi_P(\Psi) = \pi_P(\Psi \uparrow_p)$, from which the lemma follows immediately. Given any $x \in dom(\pi_P(\Psi \uparrow_p))$, for any $Q \in \mathscr{P}$, we have

$$
\begin{array}{ll}
\pi_P(\Psi \uparrow_p)(x)(Q) & \\
= (\pi_P(\Psi \uparrow_p (x)))(Q) & \text{by Def.6.13} \\
= (\Psi \uparrow_p (x))(P) & \text{by Def. 6.11} \\
= \Psi(x)(P \cup \{p\}) & \text{by Def. 6.10} \\
= \Psi(x)(P) & \text{by assumption } p \in P \\
= (\pi_P(\Psi(x)))(Q) & \text{by Def. 6.11} \\
= \pi_P(\Psi)(x)(Q) & \text{by Def. 6.13}
\end{array}
$$

Since this holds for arbitrary Q,it follows that $\pi_P(\Psi \uparrow_p) = \pi_P(\Psi)$. □

Lemma 6.19. *If $p \notin P$, then $\mu =^{l_O}_{\pi_P(\Psi)} \mu' \iff \mu =^{l_O}_{\pi_P(\Psi\downarrow_p)} \mu'$.*

Proof. Similar to the proof of Lemma 6.18. □

Lemma 6.20. *Suppose $\Psi \vdash e : t$. For $P \in \mathscr{P}$, if $t(P) \leq l_O$ and $\mu =^{l_O}_{\pi_P(\Psi)} \mu'$, $\mu \vdash e \rightsquigarrow v$ and $\mu' \vdash e \rightsquigarrow v'$, then $v = v'$.*

Proof. Consider any P which satisfies $t(P) \leq l_O$ and $\mu =^{l_O}_{\pi_P(\Psi)} \mu'$. The proof proceeds by induction on the derivation of $\Psi; A \vdash e : t$.

T-VAR We have $\Psi \vdash x : \Psi(x) = t$. Since $t(P) \leq l_O$ and $\mu =^{l_O}_{\pi_P(\Psi)} \mu'$, it is deducible that $v = \mu(x) = \mu'(x) = v'$.

T-OP We have

$$\frac{\Psi \vdash e_1 : t \qquad \Psi \vdash e_2 : t}{\Psi \vdash e_1 \circ e_2 : t}$$

and

$$\frac{\mu \vdash e_i \rightsquigarrow v_i}{\mu \vdash e_1 \circ e_2 \rightsquigarrow v_1 \circ v_2} \qquad \frac{\mu' \vdash e_i \rightsquigarrow v_i'}{\mu' \vdash e_1 \circ e_2 \rightsquigarrow v_1' \circ v_2'}$$

By induction on e_i, we can get $v_i = v_i'$. Therefore $v = v'$.

T-SUB$_e$ we have

$$\frac{\Psi \vdash e : s \qquad s \leq t}{\Psi \vdash e : t}$$

since $s(P) \leq t(P)$ and $t(P) \leq l_O$, then $s(P) \leq l_O$ as well, thus the result follows by induction on $\Psi \vdash e : s$.

□

Lemma 6.21. *Suppose* $\Psi; A \vdash c : t$. *Then for any* $P \in \mathscr{P}$, *if* $t(P) \not\leq l_O$ *and* $\mu; A; P \vdash c \rightsquigarrow \mu'$, *then* $\mu =_{\pi_P(\Psi)} \mu'$.

Proof. By induction on the derivation of $\Psi; A \vdash c : t$, with subinduction on the derivation of $\mu; A; P \vdash c \rightsquigarrow \mu'$.

T-ASS In this case $t = \Psi(x)$ and the typing derivation has the form:

$$\frac{\Psi \vdash e : \Psi(x)}{\Psi; A \vdash x := e : \Psi(x)}$$

and the evaluation under μ takes the form:

$$\frac{\mu \vdash v}{\mu; A; P \vdash x := e \rightsquigarrow \mu[x \mapsto v]}$$

That is, $\mu' = \mu[x \mapsto v]$. So μ and μ' differ possibly only in the mapping of x. Since $\Psi(x)(P) = t(P) \not\leq l_O$, that is $\pi_P(\Psi)(x) \not\leq \hat{l_O}$, the difference in the valuation of x is not observable at level l_O. It then follows from Definition 6.14 that $\mu =^{l_O}_{\pi_P(\Psi)} \mu'$.

T-ICALL In this case the command c has the form $x :=$ **Icall** $B.f(\overline{e})$ and the typing derivation takes the form:

$$\frac{\mathcal{F}t(B.f) = \overline{s} \to s' \qquad \Psi \vdash \overline{e} : \pi_{\Theta(A)}(\overline{s}) \qquad \pi_{\Theta(A)}(s') \leq \Psi(x)}{\Psi; A \vdash x := \textbf{Icall } B.f(\overline{e}) : \Psi(x)}$$

and we have that $t = \Psi(x)$. The evaluation under μ is derived as follows:

$$\frac{\mathcal{F}d(B.f) = B.f(\overline{y})\{\textbf{init } r = 0 \textbf{ in } \{c_1; \textbf{ret } r\}\} \quad \mu \vdash \overline{e} \rightsquigarrow \overline{v} \quad [\overline{y} \mapsto \overline{v}, r \mapsto 0]; B; \Theta(A) \vdash c_1 \rightsquigarrow \mu_1}{\mu; A; P \vdash x := \textbf{Icall } B.f(\overline{e}) \rightsquigarrow \mu[x \mapsto \mu_1(r)]}$$

Since $t(P) \not\leq l_O$ and $\Psi(x) = t$, we have $\Psi(x)(P) \not\leq l_O$ and therefore $\Psi(x) \not\leq l_O$ and

$$\mu =^{l_O}_{\pi_P(\Psi)} \mu[x \mapsto v'] = \mu'.$$

T-IF This follows straightforwardly from the induction hypothesis.

T-WHILE We look at the case where the condition of the while loop evaluates to true, otherwise it is trivial. In this case the typing derivation is

$$\frac{\Psi \vdash e : t \qquad \Psi; A \vdash c : t}{\Psi; A \vdash \textbf{while } e \textbf{ do } c : t}$$

and the evaluation derivation is

$$\frac{\mu \vdash e \rightsquigarrow v \quad v \neq 0 \quad \mu; A; P \vdash c \rightsquigarrow \mu_1 \quad \mu_1; A; P \vdash \textbf{while } e \textbf{ do } c \rightsquigarrow \mu'}{\mu; A; P \vdash \textbf{while } e \textbf{ do } c \rightsquigarrow \mu'}$$

Applying the induction hypothesis (on typing derivation) and the inner induction hypothesis (on the evaluation derivation) we get $\mu =^{l_O}_{\pi_P(\Psi)} \mu_1$ and $\mu_1 =^{l_O}_{\pi_P(\Psi)} \mu'$; by transitivity of $=^{l_O}_{\pi_P(\Psi)}$ we get $\mu =^{l_O}_{\pi_P(\Psi)} \mu'$.

T-SEQ This case follows from the induction hypothesis and transitivity of the indistinguishability relation.

T-LETV This follows from the induction hypothesis and the fact that we can choose fresh variables for local variables, and that the local variables are not visible outside the scope of letv.

T-CP We have:

$$\frac{\Psi \uparrow_p; A \vdash c_1 : t_1 \qquad \Psi \downarrow_p; A \vdash c_2 : t_2 \qquad t = t_1 \triangleright_p t_2}{\Psi; A \vdash \textbf{ckp}(p)\ c_1 \textbf{ else } c_2 : t}$$

There are two possible derivations for the evaluation. In one case, we have

$$\frac{p \in P \quad \mu; A; P \vdash c_1 \rightsquigarrow \mu'}{\mu; A; P \vdash \textbf{ckp}(p)\ c_1 \textbf{ else } c_2 \rightsquigarrow \mu'}$$

Since $t(P) \not\leq l_O$ and $p \in P$, by Definiton 6.12, we have $t_1(P) \not\leq l_O$. By induction hypothesis, we have $\mu =^{l_O}_{\pi_P(\Psi\uparrow_p)} \mu'$, by Lemma 6.18, we have $\mu =^{l_O}_{\pi_P(\Psi)} \mu'$.

The case where $p \notin P$ can be handled similarly, making use of Lemma 6.19.

T-SUB$_c$ Straightforward by induction.

□

Definition 6.22. A command c executed in app A is said to be *non-interferent* iff for all $\mu_1, \mu_1', \Psi, P, l_O$, if $\mu_1 =^{l_O}_{\pi_P(\Psi)} \mu_1'$, $\mu_1; A; P \vdash c \rightsquigarrow \mu_2$ and $\mu_1'; A; P \vdash c \rightsquigarrow \mu_2'$ then $\mu_2 =^{l_O}_{\pi_P(\Psi)} \mu_2'$.

Lemma 6.23. *Suppose* $\Psi; A \vdash c : t$, *for any* $P \in \mathscr{P}$, *if* $\mu_1 =^{l_O}_{\pi_P(\Psi)} \mu_1'$, $\mu_1; A; P \vdash c \rightsquigarrow \mu_2$, *and* $\mu_1'; A; P \vdash c \rightsquigarrow \mu_2'$, *then* $\mu_2 =^{l_O}_{\pi_P(\Psi)} \mu_2'$.

Proof. The proof proceeds by induction on **rank**(c), with subinduction on the derivations of $\Psi; \Theta; A \vdash c : t$ and $\mu_1; \Theta; P; A \vdash c \rightsquigarrow \mu_2$. In the following, we shall omit the superscript l_O from $=^{l_O}_{\pi_P(\Psi)}$ to simplify presentation.

T-ASS In this case, $c \equiv x := e$ and the typing derivation takes the form:

$$\frac{\Psi \vdash e : \Psi(x)}{\Psi; A \vdash x := e : \Psi(x)}$$

where $t = \Psi(x)$, and suppose the two executions of c are derived as follows:

$$\frac{\mu_1 \vdash e \rightsquigarrow v}{\mu_1; A; P \vdash x := e \rightsquigarrow \mu_1[x \mapsto v_1]}$$

$$\frac{\mu_2 \vdash e \rightsquigarrow v'}{\mu_1'; A; P \vdash x := e \rightsquigarrow \mu_1'[x \mapsto v_2]}$$

where $\mu_2 = \mu_1[x \mapsto v]$ and $\mu_2' = \mu_1'[x \mapsto v']$. Note that if $\Psi(x) \not\leq l_O$ then $\mu_2 =_{\pi_P(\Psi)} \mu_2'$ holds trivially by Definition 6.22. So let us assume $\Psi(x) \leq l_O$. Then applying Lemma 6.20 to $\mu_1 \vdash e \rightsquigarrow v$ and $\mu_1' \vdash e \rightsquigarrow v'$ we get $v = v'$, so it then follows that $\mu_2 =_{\pi_P(\Psi)} \mu_2'$.

T-IF In this case $c \equiv$ **if** e **then** c_1 **else** c_2 and we have

$$\frac{\Psi \vdash e : t \quad \Psi; A \vdash c_1 : t \quad \Psi; A \vdash c_2 : t}{\Psi; A \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : t}$$

If $t(P) \not\leq l_O$, then the lemma follows easily from Lemma 6.21. So we assume $t(P) \leq l_O$.

The evaluation derivation under μ_1 takes either one of the following forms:

$$\frac{\mu_1 \vdash e \rightsquigarrow v \quad v \neq 0 \quad \mu_1; A; P \vdash c_1 \rightsquigarrow \mu_2}{\mu_1; A; P \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow \mu_2}$$

$$\frac{\mu_1 \vdash e \rightsquigarrow v \quad v = 0 \quad \mu_1; A; P \vdash c_2 \rightsquigarrow \mu_2}{\mu; A; P \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow \mu_2}$$

We consider here only the case where $v \neq 0$; the case with $v = 0$ can be dealt with similarly. We first need to show that the evaluation of c under μ_1' would take the same if-branch. That is, suppose $\mu_1' \vdash e \rightsquigarrow v'$. Since $t(P) \leq l_O$, we can

apply Lemma 6.20 to conclude that $v = v' \neq 0$, hence the evaluation of c under μ_1' takes the form:

$$\frac{\mu_1' \vdash e \rightsquigarrow v' \quad v' \neq 0 \quad \mu_1'; A; P \vdash c_1 \rightsquigarrow \mu_2'}{\mu_1'; A; P \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow \mu_2'}$$

The lemma then follows straightforwardly from the induction hypothesis.

T-WHILE $c \equiv \textbf{while } e \textbf{ do } c_b$ and we have

$$\frac{\Psi \vdash e : t \qquad \Psi; A \vdash c : t}{\Psi; A \vdash \textbf{while } e \textbf{ do } c : t}$$

If $t(P) \not\leq l_O$, the conclusion holds by Lemma 6.21. Otherwise, since $\mu_1 =^{l_O}_{\pi_P(\Psi)} \mu_1'$, by Lemma 6.20, if $\mu_1 \vdash e \rightsquigarrow v$ and $\mu_1' \vdash e \rightsquigarrow v'$ then $v = v'$. If both are 0 then the conclusion holds according to (E-WHILE-F). Otherwise, we have

$$\frac{\mu_1 \vdash e \rightsquigarrow v \quad v \neq 0 \quad \mu_1; A; P \vdash c_b \rightsquigarrow \mu_3 \quad \mu_3; A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow \mu_2}{\mu; A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow \mu_2}$$

$$\frac{\mu_1' \vdash e \rightsquigarrow v \quad v' \neq 0 \quad \mu_1'; A; P \vdash c_b \rightsquigarrow \mu_3' \quad \mu_3'; A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow \mu_2'}{\mu; A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow \mu_2'}$$

Applying the induction hypothesis to $\Psi; A \vdash c : t$, $\mu_1; A; P \vdash c_b \rightsquigarrow \mu_3$ and $\mu_1'; A; P \vdash c_b \rightsquigarrow \mu_3'$, we obtain $\mu_3 =_{\pi_P(\Psi)} \mu_3'$. Then applying the inner induction hypothesis to $\mu_3; A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow \mu_2$ and $\mu_3'; A; P \vdash \textbf{while } e \textbf{ do } c_b \rightsquigarrow \mu_2'$, we obtain $\mu_2 =_{\pi(\Psi)P} \mu_2'$.

T-SEQ In this case we have $c \equiv c_1; c_2$ and $\Psi; A \vdash c : t$. If $t(P) \not\leq l_O$, it is a direct conclusion from Lemma 6.21; otherwise it holds by induction on c_1 and c_2.

T-LETV In this case we have $c \equiv \textbf{letv } x = e \textbf{ in } c_b$. If $t(P) \not\leq l_O$ then the lemma follows from Lemma 6.21. Otherwise, this case follows from the induction hypothesis and the fact that the mapping for the local variable x is removed in μ_2 and μ_2'.

T-ICALL In this case, c has the form $x := \textbf{Icall } B.f(\overline{e})$. Suppose the typing derivation is the following (where we label the premises for ease of reference later):

$$\frac{\mathcal{F}t(B.f) = \overline{s} \to s' \qquad (\mathbf{T_1})\ \Psi \vdash \overline{e} : \pi_{\Theta(A)}(\overline{s}) \qquad (\mathbf{T_2})\ \pi_{\Theta(A)}(s') \leq \Psi(x)}{\Psi; A \vdash x := \textbf{Icall } B.f(\overline{e}) : \Psi(x)}$$

where $t = \Psi(x)$, and the executions under μ_1 and μ_1' are derived, respectively, as follows:

$$\frac{(\mathbf{E_1})\ \mu_1 \vdash \overline{e} \rightsquigarrow \overline{v_1} \qquad (\mathbf{E_2})\ [\overline{y} \mapsto \overline{v_1}, r \mapsto 0]; B; \Theta(A) \vdash c_1 \rightsquigarrow \mu_3}{\mu_1; A; P \vdash x := \textbf{Icall } B.f(\overline{e}) \rightsquigarrow \mu_1[x \mapsto \mu_3(r)]}$$

and

$$\frac{(\mathbf{E'_1})\ \mu'_1 \vdash \overline{e} \rightsquigarrow \overline{v_2} \quad (\mathbf{E'_2})\ [\overline{y} \mapsto \overline{v_2}, r \mapsto 0]; B; \Theta(A) \vdash c_1 \rightsquigarrow \mu'_3}{\mu'_1; A; P \vdash x := \textbf{Icall}\ B.f(\overline{e}) \rightsquigarrow \mu'_1[x \mapsto \mu'_3(r)]}$$

where $\mathcal{F}d(B.f) = B.f(\overline{x})\{\textbf{init}\ r = 0\ \textbf{in}\ \{c_1; \textbf{ret}\ r\}\}$, $\mu_2 = \mu_1[x \mapsto \mu_3(r)]$ and $\mu'_2 = \mu'_1[x \mapsto \mu'_3(r)]$.

Moreover, since we consider only well-typed systems, the function $\mathcal{F}d(B.f)$ is also typeable:

$$\frac{(\mathbf{T_3})\ [\overline{y} : \overline{s}, r : s']; B \vdash c_1 : s}{\Theta \vdash B.f(\overline{y})\{\textbf{init}\ r = 0\ \textbf{in}\ \{c_1; \textbf{ret}\ r\}\} : \overline{s} \to s'}$$

First we note that if $t(P) \not\leq l_O$ then the result follows from Lemma 6.21. So in the following, we assume $t(P) \leq l_O$. Since $t = \Psi(x)$, it follows that $\Psi(x)(P) \leq l_O$. Let $\Psi' = \pi_{\Theta(A)}([\overline{y} : \overline{t}, r : s])$. We first prove several claims:

- Claim 1: $[\overline{y} \mapsto \overline{v_1}, r \mapsto 0] =_{\Psi'} [\overline{y} \mapsto \overline{v_2}, r \mapsto 0]$.

  Proof: Let $\rho = [\overline{y} \mapsto \overline{v_1}, r \mapsto 0]$ and $\rho' = [\overline{y} \mapsto \overline{v_2}, r \mapsto 0]$. We only need to check that the two mappings agree on mappings of $\overline{y}$ that are of type $\leq \hat{l}_O$. Suppose y_u is such a variable, i.e., $\Psi'(y_u) = u \leq \hat{l}_O$, and suppose $\rho(y_u) = v_u$ and $\rho'(y_u) = v'_u$ for some $y_u \in \overline{y}$. From $(\mathbf{E_1})$ we have $\mu_1 \vdash e_u \rightsquigarrow v_u$ and from $(\mathbf{E_2})$ we have $\mu'_1 \vdash e_u \rightsquigarrow v'_u$, and from $(\mathbf{T_1})$ we have $\Psi \vdash e_u : u$. Since $u \leq \hat{l}_O$, applying Lemma 6.20, we get $v_u = v'_u$.

- Claim 2: $\mu_3 =_{\Psi'} \mu'_3$.

  Proof: From Claim 1, we know that

  $$[\overline{y} \mapsto \overline{v_1}, r \mapsto 0] =_{\Psi'} [\overline{y} \mapsto \overline{v_2}, r \mapsto 0].$$

  Since $\textbf{rank}(c_1) < \textbf{rank}(c)$, we can apply the outer induction hypothesis to $(\mathbf{E_2})$, $(\mathbf{E'_2})$ and $(\mathbf{T_3})$ to obtain $\mu_3 =_{\Psi'} \mu'_3$.

- Claim 3: $\mu_3(r) = \mu'_3(r)$.

  Proof: We first note that from $(\mathbf{T_2})$ and the assumption that $\Psi(x)(P) \leq l_O$, we get $(\pi_{\Theta(A)}(s'))(P) \leq l_O$. The latter, by Definition 6.11, implies that $s'(\Theta(A)) \leq l_O$. Since $r \in dom(\Psi')$, it is obvious that $\Psi' \vdash r : s'$, $\mu_3 \vdash r \rightsquigarrow \mu_3(r)$ and $\mu'_3 \vdash r \rightsquigarrow \mu'_3(r)$. From Claim 2 above, we have $\mu_3 =_{\Psi'} \mu'_3$. Therefore by Lemma 6.20, we have $\mu_3(r) = \mu'_3(r)$.

The statement we are trying to prove, i.e., $\mu_2 =_{\pi_P(\Psi)} \mu_2'$, follows immediately from Claim 3 above.

T-CP $c \equiv \mathbf{ckp}(p)\ c_1\ \mathbf{else}\ c_2$ and we have

$$\frac{\Psi \uparrow_p; A \vdash c_1 : t_1 \quad \Psi \downarrow_p; A \vdash c_2 : t_2 \quad t = t_1 \triangleright_p t_2}{\Psi; A \vdash \mathbf{ckp}(p)\ c_1\ \mathbf{else}\ c_2 : t}$$

We need to consider two cases, one where $p \in P$ and the other where $p \notin P$.

Assume that $p \in P$. Then the evaluation of c under μ_1 and μ_1' are respectively:

$$\frac{p \in P \quad \mu_1; A; P \vdash c_1 \rightsquigarrow \mu_2}{\mu_1; A; P \vdash \mathbf{ckp}(p)\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow \mu_2}$$

and

$$\frac{p \in P \quad \mu_1'; A; P \vdash c_1 \rightsquigarrow \mu_2'}{\mu_1'; A; P \vdash \mathbf{ckp}(p)\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow \mu_2'}$$

Since $\mu_1 =_{\pi_P(\Psi)} \mu_1'$ and since $p \in P$, by Lemma 6.18, we have $\mu_1 =_{\pi_P(\Psi\uparrow_p)} \mu_1'$. Therefore by the induction hypothesis applied to $\Psi \uparrow_p; A \vdash c_1 : t_1$, $\mu_1; A; P \vdash c_1 \rightsquigarrow \mu_2$ and $\mu_1'; A; P \vdash c_1 \rightsquigarrow \mu_2'$, we obtain $\mu_2 =_{\pi_P(\Psi\uparrow_p)} \mu_2'$, and by Lemma 6.18, we get $\mu_2 =_{\pi_P(\Psi)} \mu_2'$.

For the case where $p \notin P$, we apply a similar reasoning as above, but using Lemma 6.19 in place of Lemma 6.18.

□

Definition 6.24. Let $\mathcal{S}$ be a system. A function

$$A.f(\overline{x})\{\mathbf{init}\ r = 0\ \mathbf{in}\ \{c; \mathbf{ret}\ r\}\}$$

in $\mathcal{S}$ with $FT(A.f) = \overline{t} \to t'$ is *non-interferent* if for all μ_1, μ_1', P, v, l_O, if the followings hold:

- $t'(P) \leq l_O$,
- $\mu_1 =^{l_O}_{\pi_P(\Psi)} \mu_1'$, where $\Psi = [\overline{x} : \overline{t}, r : t']$,
- $\mu_1; A; P \vdash c \rightsquigarrow \mu_2$, and $\mu_1'; A; P \vdash c \rightsquigarrow \mu_2'$,

then $\mu_2(r) = \mu_2'(r)$. The system $\mathcal{S}$ is *non-interferent* iff all functions in $\mathcal{S}$ are non-interferent.

Theorem 6.25. *Well-typed systems are non-interferent.*

6.3 Type Inference

This section describes a decidable inference algorithm for the language in §6.2.2[1]. §6.3.1 firstly rewrites the typing rules (Figure 6.2) in the form of permission trace rules (Figure 6.5), then reduces the type inference into a constraint solving problem; §6.3.2 provides procedures to solve the generated constraints.

6.3.1 Constraint Generation

6.3.1.1 Permission Tracing

In an IPC between different apps (components), there may be multiple permission checks in a calling context. Therefore, to infer a security type for an expression, a command or a function, we need to track the applications of promotions $\Psi\uparrow_p$ and demotions $\Psi\downarrow_q$ in their typing derivations. To this end, we keep the applications symbolic and collect the promotions and demotions into a sequence. In other words, we treat them as a *sequence* of promotions $\uparrow_p$ and demotions $\downarrow_p$ applied on a typing environment Ψ. For example, $(\Psi\uparrow_p)\downarrow_q$ can be viewed as an *application* of the sequence $\uparrow_p\downarrow_q$ on Ψ. The sequence of promotions and demotions is called a *permission trace* and denoted by Υ. The grammar of Υ is:

$$\Upsilon \ ::= \ \oplus p :: \Upsilon \mid \ominus p :: \Upsilon \mid \epsilon \quad p \in \mathbf{P}$$

and its length, denoted by $len(\Upsilon)$, is defined as:

$$len(\Upsilon) = \begin{cases} 0 & \text{if } \Upsilon = \epsilon \\ 1 + len(\Upsilon') & \text{if } \Upsilon = \circledcirc p :: \Upsilon', \circledcirc \in \{\oplus, \ominus\} \end{cases}$$

Definition 6.26. Given a base type t and a permission trace Υ, the *application* of Υ to t, denoted by $t \cdot \Upsilon$, is defined as:

$$t \cdot \Upsilon = \begin{cases} t & \text{if } \Upsilon = \epsilon \\ (t\uparrow_p) \cdot \Upsilon' & \text{if } \exists p, \Upsilon', s.t.\ \Upsilon = \oplus p :: \Upsilon' \\ (t\downarrow_p) \cdot \Upsilon' & \text{if } \exists p, \Upsilon', s.t.\ \Upsilon = \ominus p :: \Upsilon' \end{cases}$$

[1]Some of the definitions and proofs are excluded from this thesis since they are conducted by one collaborator; a complete technical report can be found in [156].

We also extend the application of a permission trace Υ to a typing environment Ψ (denoted by $\Psi \cdot \Upsilon$), such that $\forall x.\ (\Psi \cdot \Upsilon)(x) = \Psi(x) \cdot \Upsilon$. Based on permission traces, we give the definition of *partial subtyping relation*.

Definition 6.27. The *partial subtyping relation* $\leq_\Upsilon$, which is the subtyping relation applied on the permission trace, is defined as $s \leq_\Upsilon t$ iff. $s \cdot \Upsilon \leq t \cdot \Upsilon$.

The application of permission traces to types preserves the subtyping relation.

Lemma 6.28. $\forall s, t \in \mathcal{T},\ s \leq t \implies s \leq_\Upsilon t$ *for all* Υ.

Proof. By induction on $len(\Upsilon)$.

- $\Upsilon = \epsilon$. Since $s \cdot \Upsilon = s$ and $t \cdot \Upsilon = t$, the conclusion holds trivially.
- $\Upsilon = \oplus p :: \Upsilon'$ or $\Upsilon = \ominus p :: \Upsilon'$. Assume it is the former case, the latter is similar. By the hypothesis and Lemma 6.9, $s \uparrow_p \leq t \uparrow_p$. Then by induction, $s \uparrow_p \cdot \Upsilon' \leq t \uparrow_p \cdot \Upsilon'$, that is, $s \cdot \Upsilon \leq t \cdot \Upsilon$.

□

The following four lemmas discuss the impact of permission checking order on the same or different permissions.

Lemma 6.29. $\forall t \in \mathcal{T},\ p, q \in \mathbf{P}$ *s.t.* $p \neq q$, $t \cdot (\circledcirc p \circledast q) = t \cdot (\circledast q \circledcirc p)$, *where* $\circledcirc, \circledast, \in \{\oplus, \ominus\}$.

Proof. We only prove $t \cdot (\ominus p \oplus q) = t \cdot (\oplus q \ominus p)$, the other cases are similar. Consider any $P \in \mathscr{P}$,

$$\begin{aligned} t \cdot (\ominus p \oplus q)(P) &= ((t \downarrow_p) \uparrow_q)(P) \\ &= (t \downarrow_p (P \cup \{q\})) \\ &= t((P \cup \{q\}) \setminus \{p\}) \end{aligned}$$

and

$$\begin{aligned} t \cdot (\oplus q \ominus p)(P) &= ((t \uparrow_q) \downarrow_p)(P) \\ &= t((P \setminus \{p\}) \cup \{q\}) \\ &= t((P \cup \{q\}) \setminus (\{p\} \setminus \{q\})) \\ &= t((P \cup \{q\}) \setminus \{p\}) \end{aligned}$$

Therefore, $t \cdot (\ominus p \oplus q) = t \cdot (\oplus q \ominus p)$. □

Lemma 6.30. $\forall t \in \mathcal{T}, (t \cdot \circledcirc p) \cdot \Upsilon = (t \cdot \Upsilon) \cdot \circledcirc p$, *where* $\circledcirc \in \{\oplus, \ominus\}$ *and* $p \notin \Upsilon$.

Proof. By induction on $len(\Upsilon)$. The conclusion holds when $len(\Upsilon) = 0$ and $len(\Upsilon) = 1$ by Lemma 6.29. Suppose $len(\Upsilon) > 1$, there exists Υ' and q such that $\Upsilon = \circledast q :: \Upsilon'$ where $\circledast \in \{\oplus, \ominus\}$.

$$
\begin{array}{lll}
 & (t \cdot \circledcirc p) \cdot \Upsilon & \\
= & ((t \cdot \circledcirc p) \cdot \circledast q) \cdot \Upsilon' & \text{(by Definition 6.26)} \\
= & (t \cdot (\circledcirc p \circledast q)) \cdot \Upsilon' & \text{(by Definition 6.26)} \\
= & (t \cdot (\circledast q \circledcirc p)) \cdot \Upsilon' & \text{(by Lemma 6.29)} \\
= & ((t \cdot \circledast q) \cdot \circledcirc p) \cdot \Upsilon' & \text{(by Definition 6.26)} \\
= & ((t \cdot \circledast q) \cdot \Upsilon') \cdot \circledcirc p & \text{(induction hypothesis)} \\
= & (t \cdot (\circledast q :: \Upsilon')) \cdot \circledcirc p & \text{(by Definition 6.26)} \\
= & (t \cdot \Upsilon) \cdot \circledcirc p &
\end{array}
$$

□

Lemma 6.31. $\forall t \in \mathcal{T}$, $p \in \mathbf{P}, (t \cdot \circledcirc p) \cdot \circledast p = t \cdot (\circledcirc p)$, *where* $\circledcirc, \circledast \in \{\oplus, \ominus\}$.

Proof. By case analysis.

- $((t \cdot \oplus p) \cdot \oplus p)(P) = t(P \cup \{p\} \cup \{p\}) = t(P \cup \{p\}) = t \cdot (\oplus p)$ for each $P \in \mathscr{P}$.
- $((t \cdot \oplus p) \cdot \ominus p)(P) = t((P \setminus \{p\}) \cup \{p\}) = t(P \cup \{p\}) = t \cdot (\oplus p)$ for each $P \in \mathscr{P}$.
- $((t \cdot \ominus p) \cdot \oplus p)(P) = t((P \cup \{p\}) \setminus \{p\}) = t(P \setminus \{p\}) = t \cdot (\ominus p)$ for each $P \in \mathscr{P}$.
- $((t \cdot \ominus p) \cdot \ominus p)(P) = t((P \setminus \{p\}) \setminus \{p\}) = t(P \setminus \{p\}) = t \cdot (\ominus p)$ for each $P \in \mathscr{P}$.

□

Lemma 6.32. $\forall t \in \mathcal{T}$, $(t \cdot \Upsilon) \cdot \Upsilon = t \cdot \Upsilon$.

Proof. By induction on $len(\Upsilon)$. The conclusion holds trivially for $len(\Upsilon) = 0$ and for $len(\Upsilon) = 1$ by Lemma 6.31 . When $len(\Upsilon) > 1$, without loss of generality, assume

$\Upsilon = \oplus p :: \Upsilon'$.

$$
\begin{array}{lll}
 & t \cdot \Upsilon \cdot \Upsilon & \\
= & (t \cdot (\oplus p \cdot \Upsilon')) \cdot (\oplus p :: \Upsilon') & \\
= & (t \cdot (\Upsilon' \cdot \oplus p)) \cdot (\oplus p :: \Upsilon') & \text{(by Lemma 6.30)} \\
= & (((t \cdot \Upsilon') \cdot \oplus p) \cdot \oplus p) \cdot \Upsilon' & \text{(by Definition 6.26)} \\
= & ((t \cdot \Upsilon') \cdot \oplus p) \cdot \Upsilon' & \text{(by Lemma 6.31)} \\
= & ((t \cdot \oplus p) \cdot \Upsilon') \cdot \Upsilon' & \text{(by Lemma 6.30)} \\
= & (t \cdot \oplus p) \cdot \Upsilon' & \text{(induction hypothesis)} \\
= & t \cdot (\oplus p :: \Upsilon') & \text{(by Definition 6.26)} \\
= & t \cdot \Upsilon &
\end{array}
$$

□

Lemmas 6.29 and 6.30 state that the order of applications of promotions and demotions on *different* permissions does not affect the result. Lemmas 6.31 and 6.32 indicate that only the first application takes effect if there exist several (consecutive) applications of promotions and demotions on the *same* permission p. Therefore, we can safely keep only the first application, by removing the other applications on the same permission.

Let $occur(p, \Upsilon)$ be the number of occurrences of p in Υ. We say Υ is *consistent* iff. $occur(p, \Upsilon) \in \{0, 1\}$ for all $p \in \mathbf{P}$. In the remaining, we assume that all permission traces are consistent. Moreover, to ensure that the traces collected from the derivations of commands are consistent, we assume that in nested permission checks of a function definition, each permission is checked at most once.

Lemma 6.33. $\forall s, t \in \mathcal{T}.\ \forall p \in \mathbf{P}.(s \triangleright_p t) \cdot \Upsilon = (s \cdot \Upsilon) \triangleright_p (t \cdot \Upsilon)$, *where* $p \notin \Upsilon$.

Proof. By induction on $len(\Upsilon)$.

$len(\Upsilon) = 0$**:** Trivially.

$len(\Upsilon) > 0$**:** In this case we have $\Upsilon = \Upsilon' :: \oplus q$ or $\Upsilon' :: \ominus q$, where $len(\Upsilon') \geq 0$, $p \notin \Upsilon'$, $q \neq p$. We only prove $\Upsilon' :: \oplus q$, the other case is similar. Consider any P,

we have

$$
\begin{aligned}
& ((s \triangleright_p t) \cdot (\Upsilon' :: \oplus q))(P) \\
= \; & ((s \triangleright_p t) \cdot \Upsilon')(P \cup \{q\}) \\
= \; & ((s \cdot \Upsilon') \triangleright_p (t \cdot \Upsilon'))(P \cup \{q\}) \quad \text{(By induction)} \\
= \; & \begin{cases} (s \cdot \Upsilon')(P \cup \{q\}) & p \in P \\ (t \cdot \Upsilon')(P \cup \{q\}) & p \notin P \end{cases} \\
= \; & \begin{cases} (s \cdot (\Upsilon' :: \oplus q))(P) & p \in P \\ (t \cdot (\Upsilon' :: \oplus q))(P) & p \notin P \end{cases} \\
= \; & ((s \cdot (\Upsilon' :: \oplus q)) \triangleright_p (t \cdot (\Upsilon' :: \oplus q)))(P)
\end{aligned}
$$

□

Lemma 6.34. $\forall s, t \in \mathcal{T}.\ \forall p \in \mathbf{P}.\ s \leq t \Leftrightarrow s \cdot \oplus p \leq t \cdot \oplus p$ *and* $s \cdot \ominus p \leq t \cdot \ominus p$.

Proof. ($\Rightarrow$) by applying Lemma 6.28 with $\Upsilon = \oplus p$ and $\Upsilon = \ominus p$ respectively.
($\Leftarrow$) $\forall P \in \mathscr{P}$,

(1) If $p \in P$, by Lemma 6.8(a), $s(P) = (s \uparrow_p)(P) = (s \cdot \oplus p)(P)$ and $t(P) = (t \uparrow_p)(P) = (t \cdot \oplus p)(P)$, since $s \cdot \oplus p \leq t \cdot \oplus p$, then $s(P) \leq t(P)$.

(2) If $p \notin P$, by Lemma 6.8(b), $s(P) = (s \downarrow_p)(P) = (s \cdot \ominus p)(P)$ and $t(P) = (t \downarrow_p)(P) = (t \cdot \ominus p)(P)$, since $s \cdot \ominus p \leq t \cdot \ominus p$, then $s(P) \leq t(P)$.

This indicates that $s \leq t$. □

6.3.1.2 Permission Trace Rules

We split the applications of the promotions and demotions into two parts (i.e., typing environments and permission traces), and move the subsumption rules (guarded by permission traces) for expressions and commands to where they are needed. This yields the syntax-directed typing rules, which we call the *permission trace rules* and are given in Figure 6.5. The judgments of the trace rules are similar to those of typing rules, except that each trace rule is guarded by the permission trace Υ collected from the context, which keeps track of the adjustments of variables depending on the permission checks, and that the subtyping relation in the trace rules is the partial subtyping one $\leq_\Upsilon$.

$$\text{TT-VAR}\frac{}{\Psi;\Upsilon \vdash_t x : \Psi(x)} \qquad \text{TT-OP}\frac{\Psi;\Upsilon \vdash_t e_1 : t_1 \quad \Psi;\Upsilon \vdash_t e_2 : t_2}{\Psi;\Upsilon \vdash_t e_1 \circ e_2 : t_1 \sqcup t_2} \qquad \text{TT-ASS}\frac{\Psi;\Upsilon \vdash_t e : t \quad t \leq_\Upsilon \Psi(x)}{\Psi;\Upsilon \vdash_t x := e : \Psi(x)}$$

$$\text{TT-IF}\frac{\Psi;\Upsilon \vdash_t e : t \quad \Psi;\Upsilon;A \vdash_t c_1 : t_1 \quad \Psi;\Upsilon;A \vdash_t c_2 : t_2 \quad t \leq_\Upsilon t_1 \sqcap t_2}{\Psi;\Upsilon;A \vdash_t \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : t_1 \sqcap t_2}$$

$$\text{TT-SEQ}\frac{\Psi;\Upsilon;A \vdash_t c_1 : t_1 \quad \Psi;\Upsilon;A \vdash_t c_2 : t_2}{\Psi;\Upsilon;A \vdash_t c_1; c_2 : t_1 \sqcap t_2}$$

$$\text{TT-LETV}\frac{\Psi;\Upsilon \vdash_t e : s \quad \Psi[x : s'];\Upsilon;A \vdash_t c : t \quad s \leq_\Upsilon s'}{\Psi;\Upsilon;A \vdash_t \textbf{letv } x = e \textbf{ in } c : t}$$

$$\text{TT-WHILE}\frac{\Psi;\Upsilon \vdash_t e : s \quad \Psi;\Upsilon;A \vdash_t c : t \quad s \leq_\Upsilon t}{\Psi;\Upsilon;A \vdash_t \textbf{while } c \textbf{ do } e : t}$$

$$\text{TT-ICALL}\frac{\mathcal{F}t(B.f) = \overline{t} \to t' \quad \Psi;\Upsilon \vdash_t \overline{e} : \overline{s} \quad \overline{s} \leq_\Upsilon \overline{\pi_{\Theta(A)}(t)} \quad \pi_{\Theta(A)}(t') \leq_\Upsilon \Psi(x)}{\Psi;\Upsilon;A \vdash_t x := \textbf{Icall } B.f(\overline{e}) : \Psi(x)}$$

$$\text{TT-CP}\frac{\Psi;\Upsilon :: \oplus p;A \vdash_t c_1 : t_1 \quad \Psi;\Upsilon :: \ominus p;A \vdash_t c_2 : t_2}{\Psi;\Upsilon;A \vdash_t \textbf{ckp}(p)\ c_1 \textbf{ else } c_2 : t_1 \triangleright_p t_2} \qquad \text{TT-FUN}\frac{[\overline{x} : \overline{t}, r : t'];\epsilon;B \vdash_t c : s}{\vdash_t B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{ret } r\}\} : \overline{t} \to t'}$$

FIGURE 6.5: Permission trace rules for expressions, commands, and functions

The next two lemmas show the trace rules are *sound* and *complete* with respect to the typing rules, i.e., an expression (command, function, resp.) is typeable under the trace rules, if and only if it is typeable under the typing rules.

Lemma 6.35. *(a) If* $\Psi;\Upsilon \vdash_t e : t$*, then* $\Psi \cdot \Upsilon \vdash e : (t \cdot \Upsilon)$.
(b) If $\Psi;\Upsilon;A \vdash_t c : t$*, then* $(\Psi \cdot \Upsilon);A \vdash c : (t \cdot \Upsilon)$.
(c) If $\vdash_t B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{ret } r\}\} : \overline{t} \to t'$*, then* $\vdash B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{ret } r\}\} : \overline{t} \to t'$.

Lemma 6.36. *(a) If* $\Psi \cdot \Upsilon \vdash e : t \cdot \Upsilon$*, then there exists* s *such that* $\Psi;\Upsilon \vdash_t e : s$ *and* $s \leq_\Upsilon t$.
(b) If $(\Psi \cdot \Upsilon);A \vdash c : t \cdot \Upsilon$*, then there exists* s *such that* $\Psi;\Upsilon;A \vdash_t c : s$ *and* $t \leq_\Upsilon s$.
(c) If $\vdash B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{ret } r\}\} : \overline{t} \to s$*, then* $\vdash_t B.f(\overline{x})\{\textbf{init } r = 0 \textbf{ in } \{c; \textbf{ret } r\}\} : \overline{t} \to s$.

6.3.1.3 Constraint Generation Rules

To infer types for functions in System $\mathcal{S}$, we assign a function type $\overline{\alpha} \to \beta$ for each function $A.f$ whose type is unknown and a type variable γ for each variable x with unknown type respectively, where $\overline{\alpha}, \beta, \gamma$ are fresh type variables. Then according to permission trace rules, we try to build a derivation for each function in $\mathcal{S}$, in which we collect the side conditions (*i.e.*, the partial subtyping relation $\leq_\Upsilon$) needed by the rules. If the side conditions hold under a context, then $\mathcal{F}d(A.f)$ is typed by $\mathcal{F}t(A.f)$ under the same context for each function $A.f$ in $\mathcal{S}$.

To describe the side conditions (*i.e.*, $\leq_\Upsilon$), we define the permission guarded constraints as follows:

$$\begin{array}{lcl} c & ::= & (\Upsilon, t_l \leq t_r) \\ t_l & ::= & \alpha \mid t_g \mid t_l \sqcup t_l \mid \pi_P(t_l) \\ t_r & ::= & \alpha \mid t_g \mid t_r \sqcap t_r \mid t_r \triangleright_p t_r \mid \pi_P(t_r) \end{array}$$

where Υ is a permission trace, α is a fresh type variable and t_g is a ground type.

A *type substitution* is a finite mapping from type variables to security types: $\theta ::= \epsilon \mid \alpha \mapsto t, \theta$

Definition 6.37. Given a constraint set C and a substitution θ, we say θ is a *solution* to C, denoted by $\theta \vDash C$, iff. for each $(\Upsilon, t_l \leq t_r) \in C$, $t_l\theta \leq_\Upsilon t_r\theta$ holds.

$$\text{TG-OP}\frac{\Psi; \Upsilon \vdash_g e_1 : t_1 \rightsquigarrow C_1 \quad \Psi; \Upsilon \vdash_g e_2 : t_2 \rightsquigarrow C_2}{\Psi; \Upsilon \vdash_g e_1 \circ e_2 : t_1 \sqcup t_2 \rightsquigarrow C_1 \cup C_2}$$

$$\text{TG-ASS}\frac{\Psi; \Upsilon \vdash_g e : t \rightsquigarrow C}{\Psi; \Upsilon; A \vdash_g x := e : \Psi(x) \rightsquigarrow C \cup \{(\Upsilon, t \leq \Psi(x))\}}$$

$$\text{TG-VAR}\frac{}{\Psi; \Upsilon \vdash_g x : \Psi(x) \rightsquigarrow \emptyset}$$

$$\text{TG-LETV}\frac{\Psi; \Upsilon \vdash_g e : s \rightsquigarrow C_1 \quad \Psi[x : \alpha]; \Upsilon; A \vdash_g c : t \rightsquigarrow C_2 \quad C = C_1 \cup C_2 \cup \{(\Upsilon, s \leq \alpha)\}}{\Psi; \Upsilon; A \vdash_g \textbf{letv}\ x = e\ \textbf{in}\ c : t \rightsquigarrow C}$$

$$\text{TG-WHILE}\frac{\Psi; \Upsilon \vdash_g e : s \rightsquigarrow C \quad \Psi; \Upsilon; A \vdash_g c : t \rightsquigarrow C'}{\Psi; \Upsilon; A \vdash_g \textbf{while}\ e\ \textbf{do}\ c : t \rightsquigarrow C \cup C' \cup \{(\Upsilon, s \leq t)\}}$$

$$\text{TG-SEQ}\frac{\Psi; \Upsilon; A \vdash_g c_1 : t_1 \rightsquigarrow C_1 \quad \Psi; \Upsilon; A \vdash_g c_2 : t_2 \rightsquigarrow C_2}{\Psi; \Upsilon; A \vdash_g c_1; c_2 : t_1 \sqcap t_2 \rightsquigarrow C_1 \cup C_2}$$

$$\text{TG-IF}\frac{\Psi; \Upsilon; A \vdash_g c_1 : t_1 \rightsquigarrow C_1 \quad \Psi; \Upsilon; A \vdash_g c_2 : t_2 \rightsquigarrow C_2 \quad \Psi; \Upsilon \vdash_g e : t \rightsquigarrow C_e \quad C = C_e \cup C_1 \cup C_2 \cup \{(\Upsilon, t \leq t_1 \sqcap t_2)\}}{\Psi; \Upsilon; A \vdash_g \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : t_1 \sqcap t_2 \rightsquigarrow C}$$

$$\text{TG-CALL}\frac{\mathcal{F}t_C(B.f) = (\overline{t} \to t', C_f) \quad \Psi; \Upsilon \vdash_g \overline{e} : \overline{s} \rightsquigarrow \bigcup \overline{C_e} \quad C_a = \{(\Upsilon, \overline{s} \leq \overline{\pi_{\Theta(A)}(t)}), (\Upsilon, \pi_{\Theta(A)}(t') \leq \Psi(x))\}}{\Psi; \Upsilon; A \vdash_g x := \textbf{Icall}\ B.f(\overline{e}) : \Psi(x) \rightsquigarrow C_f \cup \bigcup \overline{C_e} \cup C_a}$$

$$\text{TG-CP}\frac{\Psi; \Upsilon :: \oplus p; A \vdash_g c_1 : t_1 \rightsquigarrow C_1 \quad \Psi; \Upsilon :: \ominus p; A \vdash_g c_2 : t_2 \rightsquigarrow C_2}{\Psi; \Upsilon; A \vdash_g \textbf{ckp}(p)\ c_1\ \textbf{else}\ c_2 : t_1 \triangleright_p t_2 \rightsquigarrow C_1 \cup C_2}$$

$$\text{TG-FUN}\frac{[\overline{x} : \overline{\alpha}, r : \beta]; \epsilon; B \vdash_g c : s \rightsquigarrow C}{\vdash_g B.f(x)\{\textbf{init}\ r = 0\ \textbf{in}\ \{c; \textbf{ret}\ r\}\} : \overline{\alpha} \to \beta \rightsquigarrow C}$$

FIGURE 6.6: Constraint generation rules for expressions, commands, and functions, with function type table $\mathcal{F}t$.

The constraint generation rules are presented in Figure 6.6, where $\mathcal{F}t_C$ is the extended function type table such that $\mathcal{F}t_C$ maps all function names to function types and their corresponding constraint sets. The judgments of the constraint rules are similar to

those of trace rules, except that each rule generates a constraint set C, which consists of the side conditions needed by the typing derivation of $\mathcal{S}$. In addition, as the function call chains starting from a command are finite, the constraint generation will terminate.

The next two lemmas show the constraint rules are *sound* and *complete* with respect to permission trace rules, i.e., the constraint set generated by the derivation of an expression (command, function, resp.) under the constraint rules is solvable, if and only if an expression (command, function, resp.) is typeable under trace rules.

Lemma 6.38. *The following statements hold:*

(a) *If* $\Psi; \Upsilon \vdash_g e : t \rightsquigarrow C$ *and* $\theta \vDash C$, *then* $\Psi\theta; \Upsilon \vdash_t e : t\theta$.

(b) *If* $\Psi; \Upsilon; A \vdash_g c : t \rightsquigarrow C$ *and* $\theta \vDash C$, *then* $\Psi\theta; \Upsilon; A \vdash_t c : t\theta$.

(c) *If* $\vdash_g B.f(x)\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{ret}}\ r\}\} : \overline{\alpha} \to \beta \rightsquigarrow C$ *and* $\theta \vDash C$, *then* $\vdash_t B.f(x)\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{ret}}\ r\}\} : \overline{\theta(\alpha)} \to \theta(\beta)$.

Lemma 6.39. *The following statements hold:*

(a) *If* $\Psi; \Upsilon \vdash_t e : t$, *then there exist* Ψ', t', C, θ *s.t.* $\Psi'; \Upsilon \vdash_g e : t' \rightsquigarrow C$, $\theta \vDash C$, $\Psi'\theta = \Psi$ *and* $t'\theta = t$.

(b) *If* $\Psi; \Upsilon; A \vdash_t c : t$, *then there exist* Ψ', t', C, θ *s.t.* $\Psi'; \Upsilon; A \vdash_g c : t' \rightsquigarrow C$, $\theta \vDash C$, $\Psi'\theta = \Psi$ *and* $t'\theta = t$.

(c) *If* $\vdash_t B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{ret}}\ r\}\} : \overline{t_p} \to t_r$, *then there exist* α, β, C, θ *s.t.* $\vdash_g B.f(\overline{x})\{\textbf{\textit{init}}\ r = 0\ \textbf{\textit{in}}\ \{c; \textbf{\textit{ret}}\ r\}\} : \overline{\alpha} \to \beta \rightsquigarrow C$, $\theta \vDash C$, *and* $(\overline{\alpha} \to \beta)\theta = \overline{t_p} \to t_r$, *where* α, β *are fresh type variables.*

Recall the function *getInfo* in Listing 6.2 and assume that *getInfo* is defined in app A (thus *A.getInfo*) and called by app B through the function *fun* (thus *B.fun*). The rephrased program is shown in Listing 6.4, where l_1, l_2 are the types for *loc* and *id* respectively, $\Theta(B) = \{q\}$, and $l_1 \sqcup l_2 = H$.

Let us apply the constraint generation rules in Figure 6.6 on each function, yielding the constraint sets C_A and C_B

$$\begin{aligned} C_A &= \{(\oplus p \oplus q, l_1 \leq \alpha), (\oplus p \ominus q, L \leq \alpha), \\ &\quad (\ominus p \oplus q, H \leq \alpha), (\ominus p \ominus q, L \leq \alpha)\} \\ C_B &= \{(\epsilon, L \leq \gamma), (\oplus p, L \leq \beta), (\ominus p, \pi_{\Theta(B)}(\alpha) \leq \gamma), \\ &\quad (\epsilon, L \leq \beta), (\epsilon, \gamma \leq \beta)\} \end{aligned}$$

```
1  A.getInfo() {
2      init r in { // Ψ(r) = α
3          ckp(p) {
4              ckp(q) r = loc; // (⊕p ⊕ q, l1 ≤ α)
5              else r = ""; // (⊕p ⊖ q, L ≤ α)
6          } else {
7              ckp(q) r = id++loc; // (⊖p ⊕ q, H ≤ α)
8              else r = ""; // (⊖p ⊖ q, L ≤ α)
9          }
10         ret r;
11     }
12 }
13 B.fun() { // B has permission q
14     init r in { // Ψ(r) = β
15         letv x = "" in { // Ψ(x) = γ, (ϵ, L ≤ γ)
16             ckp(p) r = 0;   // (⊕p, L ≤ β)
17             else x = call A.getInfo();
18             //F tC(A.getInfo) = () → α, (⊖p, πΘ(B)(α) ≤ γ)
19             if x == "" then r = 0;   // (ϵ, L ≤ β)
20             else r = 1; // (ϵ, L ≤ β), (ϵ, γ ≤ β)
21         }
22         ret r;
23     }
24 }
```

LISTING 6.4: The example in Listing 6.2 in a calling context.

and the types $t_A = () \to \alpha$ and $t_B = () \to \beta$ for the functions *getInfo* and *fun*[1] respectively. Thus, the constraint set C_{eg} for the whole program is $C_A \cup C_B$.

6.3.2 Constraint Solving

We now present an algorithm for solving the constraints generated by the rules in Figure 6.6. For these constraints, both types appearing on the two sides of subtyping are guarded by the *same* permission trace. But during the process of solving these constraints, new constraints, whose two sides of subtyping are guarded by *different* traces, may be generated. Take the constraint $(\Upsilon, \pi_P(t_l) \leq \pi_Q(\alpha))$ for example, t_l is indeed guarded by P while α is guarded by Q, where P and Q are different permission sets. So for constraint solving, we use a generalized version of the permission guarded constraints, allowing types on the two sides to be guarded by different permission traces: $((\Upsilon_l, t_l) \leq (\Upsilon_r, t_r))$ where $t_l \neq t_r$. Likewise, a *solution* to a generalized constraint set

[1]Indeed, the constraint set for *fun* is $C_A \cup C_B$, but here we focus on the constraints generated by the function itself.

C is a substitution θ, denoted by $\theta \vDash C$, such that for each $((\Upsilon_l, t_l) \leq (\Upsilon_r, t_r)) \in C$, $(t_l\theta \cdot \Upsilon_l) \leq (t_r\theta \cdot \Upsilon_r)$ holds.

It is easy to transform a permission guarded constraint set C into a generalized constraint set C': by rewriting each $(\Upsilon, t_l \leq t_r)$ as $((\Upsilon, t_l) \leq (\Upsilon, t_r))$. Moreover, it is trivial that $\theta \vDash C \iff \theta \vDash C'$. Therefore, we focus on solving generalized constraints in the following. For example, the constraint set C_{eg} can be rewritten as

$$
\begin{aligned}
&C_{eg} = \{((\epsilon, L) \leq (\epsilon, \gamma)), ((\ominus p, \pi_{\Theta(B)}(\alpha)) \leq (\ominus p, \gamma)),\\
&((\oplus p, L) \leq (\oplus p, \beta)), ((\epsilon, L) \leq (\epsilon, \beta)), ((\epsilon, \gamma) \leq (\epsilon, \beta)),\\
&((\oplus p \oplus q, l_1) \leq (\oplus p \oplus q, \alpha)), ((\oplus p \ominus q, L) \leq (\oplus p \ominus q, \alpha)),\\
&((\ominus p \oplus q, H) \leq (\ominus p \oplus q, \alpha)), ((\ominus p \ominus q, L) \leq (\ominus p \ominus q, \alpha))\}
\end{aligned}
$$

Given a permission set P and a permission trace Υ, we say P *entails* Υ, denoted by $P \vDash \Upsilon$, iff. $\forall \oplus p \in \Upsilon.\ p \in P$ and $\forall \ominus p \in \Upsilon.\ p \notin P$. A permission trace Υ is *satisfiable*, denoted by $\Delta(\Upsilon)$, iff. there exists a permission set P such that $P \vDash \Upsilon$. We write Υ_P for the permission trace that only P can entail.

A permission trace Υ can be considered as a boolean logic formula on permissions, where $\oplus$ and $\ominus$ denote positive and negative respectively, and ϵ denotes $True$. In the remaining we shall use the logic connectives on permission traces freely. We also adopt the disjunctive normal form, i.e., a disjunction of conjunctive permissions, and denote it as $dnf(\cdot)$. For example, $dnf((\oplus p) \wedge \neg(\oplus q \wedge \ominus r)) = (\oplus p \wedge \ominus q) \vee (\oplus p \wedge \oplus r)$.

The constraint solving consists of three steps: 1) *decompose* types in constraints into ground types and type variables; 2) *saturate* the constraint set by the transitivity of the subtyping relation; 3) solve the final constraint set by *merging* the lower and upper bounds of same variables and *unifying* them to emit a solution.

6.3.2.1 Decomposition

The first step is to decompose the types into the simpler ones, i.e., type variables and ground types, according to their structures. This decomposition is defined via the function dec that takes a constraint ($(\Upsilon_l, t_l, \Upsilon_r, t_r)$ for short) as input and generates a constraint set or $\perp$ (denoting unsastifiable):

$dec((\Upsilon_l, t_l, \Upsilon_r, t_r)) =$

if $t_l \cong t_l^1 \sqcup t_l^2$, **then return** $dec((\Upsilon_l, t_l^1, \Upsilon_r, t_r)) \cup dec((\Upsilon_l, t_l^2, \Upsilon_r, t_r))$

if $t_l \cong \pi_P(t)$, **then return** $dec((\Upsilon_P, t, \Upsilon_r, t_r))$

if $t_r \cong t_r^1 \sqcap t_r^2$, **then return** $dec((\Upsilon_l, t_l, \Upsilon_r, t_r^1)) \cup dec((\Upsilon_l, t_l, \Upsilon_r, t_r^2))$

if $t_r \cong \pi_P(t)$, **then return** $dec((\Upsilon_l, t_l, \Upsilon_P, t))$

if $t_r \cong t_r^1 \triangleright_p t_r^2$, **return** $dec((\Upsilon_l :: \oplus p, t_l, \Upsilon_r :: \oplus p, t_r^1))$
$\cup\, dec((\Upsilon_l :: \ominus p, t_l, \Upsilon_r :: \ominus p, t_r^2))$

if both t_l and t_r are ground, **return** $\emptyset$ if $t_l \cdot \Upsilon_l \leq t_r \cdot \Upsilon_r$ or $\bot$ otherwise

return $\{(\Upsilon_l, t_l, \Upsilon_r, t_r)\}$

After decomposition, constraints have one of the forms:

$$((\Upsilon_l, \alpha) \leq (\Upsilon_r, t_g)),\ ((\Upsilon_l, t_g) \leq (\Upsilon_r, \beta)),\ ((\Upsilon_l, \alpha) \leq (\Upsilon_r, \beta))$$

Considering the constraint set C_{eg}, only the constraint $((\ominus p, \pi_{\Theta(B)}(\alpha)) \leq (\ominus p, \gamma))$ needs to be decomposed, yielding $((\ominus p \oplus q, \alpha) \leq (\ominus p, \gamma))$.

6.3.2.2 Saturation

Considering a variable α, to ensure any lower bound (e.g., $((\Upsilon_l, t_l) \leq (\Upsilon_1, \alpha))$) is "smaller" than any of its upper bound (e.g., $((\Upsilon_2, \alpha) \leq (\Upsilon_r, t_r))$, we need to saturate the constraint set by adding these conditions. However, since our constraints are guarded by permission traces, we need to consider lower-upper bound relations only when the traces of the variable α can be entailed by the same permission set, i.e., their intersection is satisfiable. In that case, we extend the traces of both the lower and upper bound constraints such that the traces of α are the same (*i.e.*, $\Upsilon_1 \wedge \Upsilon_2$), by adding the missing traces (*i.e.*, $\Upsilon_1 \wedge \Upsilon_2 - \Upsilon_1$ for lower bound constraint while $\Upsilon_1 \wedge \Upsilon_2 - \Upsilon_2$ for the upper one, where $-$ denotes set difference). This is done by the function sat defined as:

$sat((\Upsilon_l, t_l, \Upsilon_1, \alpha), (\Upsilon_2, \alpha, \Upsilon_r, t_r)) =$

if $\Upsilon_1 \wedge \Upsilon_2$ is satisfiable, **then** let $\Upsilon_l' = \Upsilon_l \wedge (\Upsilon_1 \wedge \Upsilon_2 - \Upsilon_1)$ and $\Upsilon_r' = \Upsilon_r \wedge (\Upsilon_1 \wedge \Upsilon_2 - \Upsilon_2)$ **in** $dec((\Upsilon_l', t_l, \Upsilon_r', t_r)))$

return $\emptyset$

Assume that there is an order $<$ on type variables and the smaller variable has a higher priority. If two variables α, β with $O(\alpha) < O(\beta)$ (the orderings) are in the same constraint $\beta \leq \alpha$, we consider the larger variable β is a bound for the smaller one α,

but not vice-versa. There is a special case where both variables on two sides are the same, *e.g.*, $((\Upsilon, \alpha) \leq (\Upsilon', \alpha))$. In that case, we regroup all the trace of the variable α as the trace set $\{\Upsilon_i \mid i \in I\}$ such that the set is full (*i.e.*, $\bigvee_{i \in I} \Upsilon_i = \epsilon$) and disjoint (*i.e.*, $\forall i, j \in I. i \neq j \Rightarrow \neg\Delta(\Upsilon_i \wedge \Upsilon_j)$), and rewrite the constraints of α w.r.t. the set $\{\Upsilon_i \mid i \in I\}$. Then we treat each (Υ_i, α) as different fresh variables α_i. Therefore, there are no loops like: $(\Upsilon, \alpha) \leq \ldots \leq (\Upsilon', \alpha)$, with the ordering.

Let us consider the constraint set C_{eg} and assume that the order on variables is $O(\alpha) < O(\gamma) < O(\beta)$. There are four lower bounds and one upper bounds for α. But only the lower bound $((\ominus p \oplus q, H) \leq (\ominus p \oplus q, \alpha))$ shares the same satisfiable trace with the upper bound. So we saturate the set with the constraint $((\ominus p \oplus q, H) \leq (\ominus p, \gamma))$. Likewise, there are two lower bounds (one of which is newly generated) and one upper bound for γ. Each lower bound has a satisfiable intersected trace with the upper bound, which yields the following constraints $((\epsilon, L) \leq (\epsilon, \beta))$ and $((\ominus p \oplus q, H) \leq (\ominus p, \beta))$ (extended by $\ominus p$). While there are no upper bounds for β, so no constraints are generated. After saturation, the example set C_{eg} is

$$
\begin{aligned}
&\{((\oplus p \oplus q, l_1) \leq (\oplus p \oplus q, \alpha)), ((\oplus p \ominus q, L) \leq (\oplus p \ominus q, \alpha)),\\
&((\ominus p \oplus q, H) \leq (\ominus p \oplus q, \alpha)), ((\ominus p \ominus q, L) \leq (\ominus p \ominus q, \alpha)),\\
&((\ominus p \oplus q, \alpha) \leq (\ominus p, \gamma)), ((\epsilon, L) \leq (\epsilon, \gamma)),\\
&((\ominus p \oplus q, H) \leq (\ominus p, \gamma)), ((\epsilon, \gamma) \leq (\epsilon, \beta)),\\
&((\oplus p, L) \leq (\oplus p, \beta)), ((\epsilon, L) \leq (\epsilon, \beta)), ((\ominus p \oplus q, H) \leq (\ominus p, \beta))\}
\end{aligned}
$$

Lemma 6.40. *Given two types s, t and a permission trace Υ, then $s \leq t \iff s \cdot \Upsilon \leq t \cdot \Upsilon$ and $\forall \Upsilon' \in dnf(\neg\Upsilon).\ s \cdot \Upsilon' \leq t \cdot \Upsilon'$.*

Lemma 6.41. *If $C \rightsquigarrow_r C'$, then $C \vDash C'$ and $C' \vDash C$, where $r \in \{d, s, m\}$.*

6.3.2.3 Unification

Since our constraints are guarded by permission traces, we need to consider the satisfiability of (any subset of) the permission traces of a variable α under any permission set when constructing a type for it. Let us consider a variable α and assume that the constraints on it to be solved are $\{((\Upsilon_i^l, t_i^l) \leq (\Upsilon_i, \alpha))\}_{i \in I}$ (*i.e.*, the lower bounds) and $\{((\Upsilon_j, \alpha) \leq (\Upsilon_j^r, t_j^r))\}_{j \in J}$ (*i.e.*, the upper bounds). This indicates that under a permission set P, α can take such a type t that is bigger than $t_i^l \cdot \Upsilon_i^l$ if $P \vDash \Upsilon_i$ and is smaller

than $t_j^r \cdot \Upsilon_j^r$ if $P \vDash \Upsilon_j$. Consequently, t should be bigger than the union $\bigsqcup_{i \in I'} t_i^l \cdot \Upsilon_i^l$ and smaller than the intersection $\bigsqcap_{j \in J'} t_j^r \cdot \Upsilon_j^r$ if all the traces $\Upsilon_i \in I'$ and $\Upsilon_j \in J'$ are entailed by P. In other words, α can take any type ranging from $\bigsqcup_{i \in I'} t_i^l \cdot (\Upsilon_i^l \wedge \Upsilon_i')$ to $\bigsqcap_{j \in J'} t_j^r \cdot (\Upsilon_j^r \wedge \Upsilon_j')$ if only the intersection trace $\bigwedge_{i \in I'} \Upsilon_i \wedge \bigwedge_{j \in J'} \Upsilon_j$ is satisfiable, which indicates that α is equivalent to $(\bigsqcup_{i \in I'} t_i^l \cdot (\Upsilon_i^l \wedge \Upsilon_i') \sqcup \alpha') \sqcap \bigsqcap_{j \in J'} t_j^r \cdot (\Upsilon_j^r \wedge \Upsilon_j')$, where Υ_i' and Υ_j' are the missing traces to extend Υ_i and Υ_j to the intersection trace respectively, and α' is a fresh variable. The type above is exactly what we want. We define the construction of the type above via the function $merge$:

$merge(\{(\Upsilon_i^l, t_i^l, \Upsilon_i, \alpha)\}_{i \in I}, \{(\Upsilon_j, \alpha, \Upsilon_j^r, t_j^r)\}_{j \in J}) =$
  let $\phi(I', J') = \{\Upsilon \in dnf(\bigwedge_{i \in I'} \Upsilon_i \wedge \bigwedge_{j \in J'} \Upsilon_j \wedge \bigwedge_{i \in I \setminus I'} \neg \Upsilon_i \wedge \bigwedge_{j \in J \setminus J'} \neg \Upsilon_j) \mid \Delta(\Upsilon)\}$ **in**
  let $t^{\sqcup}_{I',\Upsilon} = \bigsqcup_{i \in I'} t_i^l \cdot (\Upsilon_i^l \wedge (\Upsilon - \Upsilon_i))$ **in**
  let $t^{\sqcap}_{J',\Upsilon} = \bigsqcap_{j \in J'} t_j^r \cdot (\Upsilon_j^r \wedge (\Upsilon - \Upsilon_j))$ **in**
  $\{\Upsilon \mapsto (t^{\sqcup}_{I',\Upsilon} \sqcup \alpha_\Upsilon) \sqcap t^{\sqcap}_{J',\Upsilon}\}_{I' \subseteq I, J' \subseteq J, \Upsilon \in \phi(I', J')}$

(with the convention $t^{\sqcup}_{\emptyset,\Upsilon} = L$ and $t^{\sqcap}_{\emptyset,\Upsilon} = H$.)

Moreover, due to the absence of loops in constraints and that the variables are in order, we can solve the constraints in reverse order on variables by unification. The unification algorithm $unify$ is presented as follows.

$unify(C)$ =
  let $subst\ \theta\ ((\Upsilon_l, t_l, \Upsilon_r, t_r)) = ((\Upsilon_l, t_l\theta, \Upsilon_r, t_r\theta))$ **in**
  select $\{(\Upsilon_i^l, t_i^l, \Upsilon_i, \alpha)\}_{i \in I}$ and $\{(\Upsilon_j, \alpha, \Upsilon_j^r, t_j^r)\}_{j \in J}$ for the maximum variable α **if** exists, $merge$ them as t_α
    let C' be the remaining constraints **in**
    let $C'' = List.map\ (subst\ [\alpha \mapsto t_\alpha])\ C'$ **in**
    let $\theta' = unify(C'')$ **in** $\theta'[\alpha \mapsto t_\alpha]$
  else return []

Let us consider the constraint set C_{eg} again and take the constraints on the maximum variable β, which are the following set without any upper bounds

$$\{((\oplus p, L) \leq (\oplus p, \beta)), ((\epsilon, L) \leq (\epsilon, \beta)), ((\ominus p \oplus q, H) \leq (\ominus p, \beta))\}$$

By applying the function $merge$, we construct for β the type $t_\beta = \{\oplus p \mapsto (L \sqcup \beta') \sqcap H, \ominus p \mapsto H\}$, where only the common traces (*i.e.*, $\oplus p$ and $\ominus p$) for the subsets $\{\epsilon, \oplus p\}$ and $\{\epsilon, \ominus p\}$ are satisfiable, and β' is a fresh variable. For simplicity, we pick the least upper bound as possible when constructing types. So we take $\{\oplus p \mapsto L, \ominus p \mapsto H\}$ as t_β instead. Next, we substitute t_β for all the occurrences of β in the remaining

constraints and continue with the constraints on γ and α. Finally, the types constructed for γ and α are $t_\gamma = t_\beta$ and $t_\alpha = \{\oplus p \oplus q \mapsto l_1, \oplus p \ominus q \mapsto L, \ominus p \oplus q \mapsto H, \ominus p \ominus q \mapsto L\}$, respectively. Therefore, the types we infer for *A.getInfo* and *B.fun* are $() \rightarrow t_\alpha$ and $() \rightarrow t_\beta$, respectively.

Lemma 6.42. *If* $unify(C) = \theta$*, then* $\theta \vDash C$.

Lemma 6.43. *If* $\theta \vDash C$*, then there exist* θ' *and* θ'' *such that* $unify(C) = \theta'$ *and* $\theta = \theta'\theta''$.

Let sol be the function for the constraint solving algorithm, i.e., $sol(C) = unify(sat(dec(C)))$. It is provable that this algorithm is sound and complete.

Lemma 6.44. *If* $sol(C) = \theta$*, then* $\theta \vDash C$.

Lemma 6.45. *If* $\theta \vDash C$*, then there exist* θ' *and* θ'' *such that* $sol(C) = \theta'$ *and* $\theta = \theta'\theta''$.

To conclude, an expression (command, function, resp.) is typeable, iff it is derivable under the constraint rules with a solvable constraint set by our algorithm. Therefore, our type inference system is sound and complete. Moreover, as the function call chains are finite, the constraint generation terminates with a finite constraint set, which can be solved by our algorithm in finite steps. Thus, our type inference system terminates.

Theorem 6.46. *The type inference system is sound, complete and decidable.*

6.4 Related Work

We discuss the related work on the type checking based verification on information flow security for the Android platform.

We have discussed extensively the work by Banerjee and Naumann [148] and highlights the major differences between my work and theirs in §6.1.

Flow-sensitive and value-dependent information flow type systems provide a general treatment of security types that may depend on other program variables or execution contexts [157–175]. Hunt and Sands [175] proposed a flow-sensitive type system where order of execution is taken into account in the analysis, and demonstrated that

the system is precise but can be simply described. Mantel et. al. [169] introduced a rely-guarantee style reasoning for information flow security in which the same variable can be assigned different security labels depending on whether some assumption is guaranteed, which is similar to our notion of permission-dependent security types. Li and Zhang [174] proposed both flow-sensitive and path-sensitive information flow analysis with program transformation techniques and dependent types. Information flow type systems that may depend on execution contexts have been considered in work on program synthesis [158] and dynamic information flow control [168]. Our permission context can be seen as a special instance of execution context, however, our intended applications and settings are different from [158, 168], and issues such as parameter laundering does not occur in their setting. Lourenço and Caires [165] provided a precise dependent type system where security labels can be indexed by data structures, which can be used to encode the dependency of security labels on other values in the system. It may be possible to encode our notion of security types as a dependent type in their setting, by treating permission sets explicitly as an additional parameter to a function or a service, and to specify security labels of the output of the function as a type dependent on that parameter. It remains unclear to us how one could give a general construction of the index types in their type system that would correspond to our security types, and how the merge operator would translate to their dependent type constructors, among other things.

Recent research on information flow has also been conducted to deal with Android security issues ([13, 143, 152, 176–179]). SCandroid [143, 179] is a tool automating security certification of Android apps that focuses on typing communication between apps. Unlike my work, they do not consider implicit flows, and do not take into account access control in their type system. Ernst et al [13] proposed a verification model, SPARTA, for use in app stores to guarantee that apps are free of malicious information flows. Their approach requires the collaboration between software vendor and app store auditor and the additional modification of Android permission model to fit for their Information Flow Type-checker; soundness proof is also absent. My work is done in the context of providing information flow security *certificates* for Android apps, following the Proof-Carrying-Code architecture by Necula and Lee [180] and does not require

extra changes on existing Android application supply chain systems.

6.5 Conclusion

We have provided a lightweight yet precise type system featuring Android permission model that enforces secure information flow in an imperative language and proved its soundness in terms of non-interference. Compared to existing work, our type system can specify a broader range of security policies, including non-monotonic ones. We have also proposed a decidable type inference algorithm by reducing it to a constraint solving problem.

Chapter 7

Discussion and Conclusion

7.1 Discussion

It is observable that greybox fuzzing is quite effective in revealing implementation vulnerabilities caused by application crashes in a wide range of C or C++ programs. In fact, most of the binary CVEs assigned in the latest four years are found by fuzzers. In particular, FOT and its extensions HAWKEYE and DOUBLADE have been able to discover more than 200 security bugs (c.f., Appendix B), among these 51 CVE IDs have been assigned. Despite that, it is worth noting that fuzzing itself almost only detects implementation relevant vulnerabilities caused by crashes which is rather specific to the applied program languages. Indeed, with some argumentation, fuzzing is able to detect certain logic errors by applying some differential fuzzing techniques [38]. However, the determination of the vulnerabilities still relies on the observations of crash behaviors that may or may not be artificially instrumented. As a result, fuzzing only copes with the very concrete issues occur in certain software. Moreover, Fuzzing cannot guarantee that the underlying programs are free of bugs as it essentially knows nothing about the actual coverage.

Quite the contrary, verification usually relies on top-down modeling of the vulnerability behaviors and rigor proofs. The merits are that verification guarantees that certain security property holds under all circumstances, and are not restricted specific software. However, the problem of verification is that the precondition required for proving is

usually quite strict, greatly reducing the practicality of the approach. With the increasing complexity of the software, it is typically impractical to apply the verification by considering all the security-relevant properties. Fortunately, it is possible to abstract the interested these properties with the assistance of some manual efforts.

In this sense, I believe that there is no silver bullet: neither of testing or verification is perfect in securing the software systems. It is however possible to combine them by utilizing their strong points and mitigating their weaknesses.

7.2 Conclusion

Testing and verification are two effective solutions to securing the software systems. In this thesis, we applied the greybox fuzz testing on the programs that may suffer from low level implementation vulnerabilities. To fulfill this, we implemented our fuzzing framework, FOT, which features configurability and extensibility to cater for various fuzzing scenarios. Based on FOT, we proposed two extensions that particularly handle two fuzzing scenarios: we applied HAWKEYE to handle the directedness guidance during fuzzing, and DOUBLADE to improve the efficiency of fuzzing on multi-threaded programs. Additionally, we applied the security type system verification to enforce the non-interference property in the Android like systems, which guarantees that the underlying system is free of information leakage as long as it can be well-typed. Undoubtedly, both of these two approaches have their own limitations, but they are complementary to each other in the sense that they safeguard the underlying systems with multiple levels of security requirements. In the future, we plan to integrate them to provide a comprehensive solution that can be used to reveal different aspects of vulnerabilities.

Appendix A

List of Publications

1. **Hongxu Chen**, Yuekang Li, Bihuan Chen, Yinxing Xue, Yang Liu, "FOT: A Versatile, Configurable, Extensible Fuzzing Framework", ESEC/FSE 2018 Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Pages 867-870, http://doi.acm.org/10.1145/3236024.3264593, DOI: 10.1145/3236024.3264593. [2]

2. **Hongxu Chen**, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed greybox fuzzing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, pages 2095–2108, 2018. [1]

3. **Hongxu Chen**, Alwen Tiu, Zhiwu Xu, and Yang Liu. A permission-dependent type system for secure information flow analysis. In 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018, pages 218–232, 2018. [3]

4. Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, **Hongxu Chen**, Jun Sun, and Jie Zhang. Auditing anti-malware tools by evolving android malware and dynamic loading technique. IEEE Transactions Information Forensics and Security, 12(7):1529–1544, 2017. [185]

5. Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and **Hongxu Chen**. S-looper: automatic summarization for multipath string loops. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, pages 188–198, 2015. [58]

Appendix B

List of Discovered Bugs

Projects	Selected Discovered Bugs/Vulnerabilities
binaryen	17 bugs
CImg	2 bugs
Espruino	CVE-2018-11590, CVE-2018-11591, CVE-2018-11592, CVE-2018-11593, CVE-2018-11594, CVE-2018-11595, CVE-2018-11596, CVE-2018-11597, CVE-2018-11598
Exiv2	CVE-2018-19107, CVE-2018-19108, CVE-2018-19535
FFmpeg	CVE-2018-15822, CVE-2018-14394, CVE-2018-14395
FLIF	2 bugs
glibc	CVE-2009-5155, CVE-2019-9169, CVE-2018-20796
GNU bc	18 bugs
GNU Binutils	CVE-2019-11473, CVE-2018-17794
GNU diffutils	2 bugs
GNU grep	1 bug
GNU sed	2 bugs
GraphicsMagick	1 performance issue, CVE-2019-11474
GPAC	15 bugs
JSMN	1 bug
ImageMagick	CVE-2018-14560, CVE-2018-14561, CVE-2019-11470, CVE-2019-11472

Intel XED	2 categories of bugs
libheif	CVE-2019-11471
libjpeg-turbo	CVE-2018-14498
libtoml	6 bugs
lepton	4 bugs, CVE-2018-20819, CVE-2018-20820
liblnk	19 bugs
liblouis	1 bug
libmobi	4 bugs
libpff	3 bugs
libsass	10 bugs, CVE-2018-19837, CVE-2018-19838, CVE-2018-19839, CVE-2018-20821, CVE-2018-20822
libsixel	5 crashes and 1 performance issue
libvips	11 bugs
libvpx	CVE-2019-11475
MJS	33 bugs
mozjpeg	1 bug
mxml	CVE-2018-20592, CVE-2018-20593
openh264	1 bug
poppler	CVE-2018-20662
radare2	40+ bugs, CVE-2018-19842, CVE-2018-19843, CVE-2018-20455, CVE-2018-20456, CVE-2018-20457, CVE-2018-20458, CVE-2018-20459, CVE-2018-20460, CVE-2018-20461
solidity	2 crashes and 1 performance issue
Swift	6 bugs (SR-8467, SR-8468, SR-8476, SR-8483, SR-8576, SR-8577)
tinyexr	6 bugs
WAVM	2 bugs (CVE-2018-17292, CVE-2018-17293)
WavPack	CVE-2018-19840, CVE-2018-19841
WebM	1 bug
xdrpp	1 bug
yaml-cpp	2 bugs

Bibliography

[1] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *CCS '18*. New York, NY, USA: ACM, 2018, pp. 2095–2108. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243849

[2] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu, "FOT: A Versatile, Configurable, Extensible Fuzzing Framework," in *ESEC/FSE '18*. ACM Press, 11 2018, pp. 867–870.

[3] H. Chen, A. Tiu, Z. Xu, and Y. Liu, "A permission-dependent type system for secure information flow analysis," in *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, 2018, pp. 218–232. [Online]. Available: https://doi.org/10.1109/CSF.2018.00023

[4] Synopsys. (2017) The heartbleed bug. [Online]. Available: http://heartbleed.com/

[5] Ethiopian Airlines Group. (2019) Aircraft Accident Investigation Bureau Preliminary Report. [Online]. Available: https://flightsafety.org/wp-content/uploads/2019/04/Preliminary-Report-B737-800MAX-ET-AVJ.pdf

[6] Flexera. (2018) Vulnerability review 2018 global trends. [Online]. Available: https://www.flexera.com/media/pdfs/research-svm-vulnerability-review-2018.pdf

[7] G. Meng, Y. Liu, J. Zhang, A. Pokluda, and R. Boutaba, "Collaborative security: A survey and taxonomy," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 1:1–1:42, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2785733

[8] A. Narayanan, L. Yang, L. Chen, and L. Jinliang, "Adaptive and scalable android malware detection through online learning," in *2016 International Joint Conference on Neural Networks (IJCNN)*, July 2016, pp. 2484–2491.

[9] B. Hailpern and P. Santhanam, "Software Debugging, Testing, and Verification," *IBM Syst. J.*, vol. 41, no. 1, pp. 4–12, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1147/sj.411.0004

[10] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, "Model-based security testing: A taxonomy and systematic classification," *Softw. Test. Verif. Reliab.*, vol. 26, no. 2, pp. 119–148, Mar. 2016. [Online]. Available: http://dx.doi.org/10.1002/stvr.1580

[11] A. Armando, G. Pellegrino, R. Carbone, A. Merlo, and D. Balzarotti, "From model-checking to automated testing of security protocols: Bridging the gap," in *Tests and Proofs*, A. D. Brucker and J. Julliand, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–18.

[12] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security and Privacy*, vol. 7, no. 1, pp. 50–57, Jan. 2009.

[13] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *CCS '14*. New York, NY, USA: ACM, 2014, pp. 1092–1104.

[14] C. Smith, "Google isn't fixing a serious android security flaw for months." [Online]. Available: http://bgr.com/2017/05/09/android-permissions-security-flaw-android-o/

[15] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing As Markov Chain," in *CCS '16*. New York, NY, USA: ACM Press, 2016, pp. 1032–1043.

[16] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *CCS '17*. New York, NY, USA: ACM Press, 2017, pp. 2329–2344.

[17] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *NDSS '19*. San Diego, CA USA: The Internet Society, 2019, pp. 1–15.

[18] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *SP '18*. San Francisco, California, USA: IEEE Press, 2018, pp. 1–12.

[19] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," *CoRR*, vol. abs/1803.01307, pp. 711–725, 2018. [Online]. Available: https://arxiv.org/abs/1803.01307v2

[20] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, Science, and Engineering," *CoRR*, vol. abs/1812.00140, pp. 1–29, 2018. [Online]. Available: http://arxiv.org/abs/1812.00140

[21] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-Driven Seed Generation for Fuzzing," in *SP '17*, May 2017, pp. 579–594.

[22] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state Based Binary Fuzzing," in *ESEC/FSE '17*. New York, NY, USA: ACM Press, 2017, pp. 627–637.

[23] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing," *CoRR*, vol. abs/1812.01197, pp. 1–12, 2018.

[24] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *ICSE '20*, 2020.

[25] H. Wang, X. Xie, Y. Li, C. Wen, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *ICSE '20*, 2020.

[26] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[27] M. Zalewski. (2014) American Fuzzy Lop. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[28] LLVM. (2015) libFuzzer. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[29] Google. (2010) honggfuzz. [Online]. Available: https://github.com/google/honggfuzz

[30] Google. (2019) ClusterFuzz. [Online]. Available: https://google.github.io/clusterfuzz/

[31] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *PLDI '05*. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[32] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *OSDI '08*, 2008, pp. 209–224.

[33] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[34] I. Free Software Foundation. (1997) GCC, the GNU Compiler Collection. [Online]. Available: https://gcc.gnu.org/

[35] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *PLDI '07*. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[36] O. Levi. (2018) Pin - A Dynamic Binary Instrumentation Tool. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[37] C. Lemieux and K. Sen, "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage," in *ASE 2018*. New York, NY, USA: ACM, 2018, pp. 475–485. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238176

[38] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: efficient domain-independent differential testing," in *S&P '17*, 2017, pp. 615–632.

[39] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 146–157. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330579

[40] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 2019, pp. 477–487. [Online]. Available: https://doi.org/10.1145/3338906.3338954

[41] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[42] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[43] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *SOSP 1982*, 1982, pp. 11–20.

[44] D. Bell and L. LaPadula, "Secure computer system: Mathematical foundations and model," Technical Report M74-244, 1973.

[45] Cesanta Software. (2016) mjs. [Online]. Available: https://github.com/cesanta/mjs

[46] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: Generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 679–690. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238219

[47] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Semantic slicing of software version histories," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 182–201, Feb 2018.

[48] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 678–689. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950350

[49] "Automatic hot patch generation for android kernels," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/xu

[50] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[51] J.-Y. Audibert, R. Munos, and C. Szepesvári, "Exploration-exploitation tradeoff using variance estimates in multi-armed bandits," *Theoretical Computer Science*, vol. 410, no. 19, pp. 1876 – 1902, 2009, algorithmic Learning Theory. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S030439750900067X

[52] K. Serebryany and M. Böhme. (2017) AFLGo: Directing AFL to reach specific target locations. [Online]. Available: https://groups.google.com/forum/#!topic/afl-users/qcqFMJa2yn4

[53] K. Kosako. (2002) Oniguruma. [Online]. Available: https://github.com/kkos/oniguruma

[54] Agostino Sarubbo. (2017) binutils: NULL pointer dereference in concat_filename (dwarf2.c) (INCOMPLET FIX FOR CVE-2017-15023). [Online]. Available: https://blogs.gentoo.org/ago/2017/10/24/binutils-null-pointer-dereference-in-concat_filename-dwarf2-c-incomplete-fix-for-cve-2017-15023

[55] A. Sarubbo. (2017) binutils: NULL pointer dereference in concat_filename (dwarf2.c). [Online]. Available: https://blogs.gentoo.org/ago/2017/10/03/binutils-null-pointer-dereference-in-concat_filename-dwarf2-c

[56] D. Britz. (2014) Exploitation vs Exploration. [Online]. Available: https://medium.com/@dennybritz/exploration-vs-exploitation-f46af4cf62fe

[57] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language," DIKU, University of Copenhagen, Tech. Rep., 1994.

[58] X. Xie, Y. Liu, W. Le, X. Li, and H. Chen, "S-looper: Automatic summarization for multipath string loops," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 188–198. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771815

[59] J. Li, J. Sun, L. Li, Q. L. Le, and S.-W. Lin, "Automatic loop-invariant generation and refinement through selective sampling," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 782–792. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155660

[60] X. Xie, B. Chen, L. Zou, S.-W. Lin, Y. Liu, and X. Li, "Loopster: Static loop termination analysis," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 84–94. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106260

[61] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: computing disjunctive loop summary via path dependency analysis," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 61–72.

[62] X. Xie, B. Chen, L. Zou, Y. Liu, W. Le, and X. Li, "Automatic loop summarization via path dependency analysis," *IEEE Transactions on Software Engineering*, 2018.

[63] S.-W. Lin, J. Sun, H. Xiao, Y. Liu, D. Sanán, and H. Hansen, "FiB: Squeezing Loop Invariants by Interpolation Between Forward/Backward Predicate Transformers," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 793–803. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155661

[64] M. Zalewski. (2014) Technical "whitepaper" for afl-fuzz. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt

[65] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-flow Analysis in LLVM," in *CC '16*. New York, NY, USA: ACM Press, 2016, pp. 265–266.

[66] G. Binutils. (1990) GNU Binutils. [Online]. Available: https://www.gnu.org/software/binutils/

[67] PHP. (1994) PHP: Hypertext Preprocessor. [Online]. Available: http://www.php.net/

[68] Google. (2017) Fuzzer Test Suite. [Online]. Available: https://github.com/google/fuzzer-test-suite

[69] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J.-P. Seifert, and A. Feldmann, "Static program analysis as a fuzzing aid," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds. Springer International Publishing, 2017, pp. 26–47.

[70] M. Zalewski. (2016) "FidgetyAFL" implemented in 2.31b. [Online]. Available: https://groups.google.com/forum/#!topic/afl-users/1PmKJC-EKZ0

[71] AFLGo. (2018) GitHub - AFLGo. [Online]. Available: https://github.com/aflgo/aflgo/issues

[72] A. Vargha, A. Vargha, and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[73] fuzzer-test suite. (2018) libpng-1.2.56/test-libfuzzer.sh. [Online]. Available: https://github.com/google/fuzzer-test-suite/blob/master/libpng-1.2.56/test-libfuzzer.sh

[74] LLVM/Clang. (2013) Clang Static Analyzer. [Online]. Available: https://clang-analyzer.llvm.org/

[75] W. Wang, H. Sun, and Q. Zeng, "SeededFuzz: Selecting and Generating Seeds for Directed Fuzzing," in *TASE '16*, 07 2016, pp. 49–56.

[76] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS Binary Search," in *FSE '16*. New York, NY, USA: ACM Press, 2016, pp. 678–689.

[77] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation," *IEEE Trans Software Engineering*, p. (to appear), 2018.

[78] Hex-Rays. (2013) IDA. [Online]. Available: https://www.hex-rays.com/index.shtml

[79] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed Symbolic Execution," in *SAS'11*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 95–111.

[80] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *SEC '13*. Berkeley, CA, USA: USENIX Association, 2013, pp. 49–64.

[81] W. Jin and A. Orso, "BugRedux: Reproducing Field Failures for In-house Debugging," in *ICSE '12*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 474–484.

[82] P. D. Marinescu and C. Cadar, "KATCH: High-coverage Testing of Software Patches," in *ESEC/FSE 2013*. New York, NY, USA: ACM Press, 2013, pp. 235–245.

[83] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *NDSS '16*, 2016, pp. 1–16.

[84] V. Ganesh, T. Leek, and M. Rinard, "Taint-based Directed Whitebox Fuzzing," in *ICSE '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 474–484.

[85] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *NDSS '17*, Feb. 2017, pp. 1–14.

[86] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *SP '10*, no. 16, May 2010, pp. 497–512.

[87] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," in *CCS '17*. New York, NY, USA: ACM Press, 2017, pp. 2313–2328.

[88] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by Program Transformation," in *SP '18*, 2018, pp. 697–710.

[89] S. Hong and M. Kim, "A Survey of Race Bug Detection Techniques for Multithreaded Programmes," *Softw. Test. Verif. Reliab.*, vol. 25, no. 3, pp. 191–217, May 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1564

[90] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *CCS '18*. New York, NY, USA: ACM, 2018, pp. 2123–2138. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243804

[91] IEEE and T. O. Group, "POSIX.1-2017," 2001. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/

[92] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed. San Francisco, CA, USA: No Starch Press, 2010.

[93] P. Di and Y. Sui, "Accelerating Dynamic Data Race Detection Using Static Thread Interference Analysis," in *PMAM@PPoPP 2016*. New York, NY, USA: ACM, 2016, pp. 30–39. [Online]. Available: http://doi.acm.org/10.1145/2883404.2883405

[94] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *CGO 2016*. New York, NY, USA: ACM, 2016, pp. 160–170. [Online]. Available: https://doi.org/10.1145/2854038.2854043

[95] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[96] I. Chowdhury and M. Zulkernine, "Using Complexity, Coupling, and Cohesion Metrics As Early Indicators of Vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, Mar. 2011. [Online]. Available: http://dx.doi.org.ezlibproxy1.ntu.edu.sg/10.1016/j.sysarc.2010.06.003

[97] M. O. Shudrak and V. Zolotarev, "Improving Fuzzing Using Software Complexity Metrics," *CoRR*, vol. abs/1807.01838, pp. 1–16, 2018.

[98] H. Chen. (2019) Doublade. [Online]. Available: https://sites.google.com/view/mtfuzz

[99] OpenMP. (1997) OpenMP. [Online]. Available: https://www.openmp.org/

[100] N. Willis. (2014) Race detection and more with ThreadSanitizer 2. [Online]. Available: https://lwn.net/Articles/598486/

[101] 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 *NDSS 2016*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[102] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *WBIA '09*. New York, NY, USA: ACM, 2009, pp. 62–71. [Online]. Available: http://doi.acm.org/10.1145/1791194.1791203

[103] M. Izdebski. (2011) lbzip2 - parallel bzip2 compression utility. [Online]. Available: http://lbzip2.org/

[104] J. Huang, "UFO: predictive concurrency use-after-free detection," in *ICSE 2018*, 2018, pp. 609–619.

[105] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 320–331, 2006.

[106] V. Vojdani and V. Vene, "Goblint: Path-sensitive data race analysis," *Annales Univ. Sci. Budapest., Sect. Comp.*, pp. 1–12, 2009.

[107] B. Liu and J. Huang, "D4: Fast Concurrency Debugging with Parallel Differential Analysis," in *PLDI 2018*. New York, NY, USA: ACM, 2018, pp. 359–373. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192390

[108] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "RacerD: Compositional Static Race Detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 144:1–144:28, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276514

[109] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *PLDI 2009*. New York, NY, USA: ACM, 2009, pp. 121–133. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542490

[110] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265927

[111] G. Jin, A. V. Thakur, B. Liblit, and S. Lu, "Instrumentation and sampling strategies for cooperative concurrency bug isolation," in *OOPSLA 2010*, 2010, pp. 241–255.

[112] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: a coverage-driven testing tool for multithreaded programs," in *OOPSLA 2012*, 2012, pp. 485–502.

[113] Valgrind. (2000) Helgrind: a thread error detector. [Online]. Available: http://valgrind.org/docs/manual/hg-manual.html

[114] K. Sen, "Effective random testing of concurrent programs," in *ASE '07*, 2007, pp. 323–332.

[115] ——, "Race directed random testing of concurrent programs," in *PLDI '08*, 2008, pp. 11–21.

[116] P. Joshi, C. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *PLDI 2009*, 2009, pp. 110–120.

[117] C.-S. Park and K. Sen, "Randomized Active Atomicity Violation Detection in Concurrent Programs," in *ESEC/FSE '08*. New York, NY, USA: ACM, 2008, pp. 135–145. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453121

[118] Y. Cai and W. K. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," in *ICSE 2012*, 2012, pp. 606–616.

[119] V. Terragni, S. Cheung, and C. Zhang, "RECONTEST: Effective Regression Testing of Concurrent Programs," in *ICSE 2015*, 2015, pp. 246–256.

[120] T. Yu, Z. Huang, and C. Wang, "ConTesa: Directed Test Suite Augmentation for Concurrent Software," *IEEE Transactions on Software Engineering*, 07 2018.

[121] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL 2005*, 2005, pp. 110–121.

[122] A. Zaks and R. Joshi, "Verifying Multi-threaded C programs with SPIN," in *SPIN '08*, 2008, pp. 325–342.

[123] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software," in *SPIN '07*, 2007, pp. 58–75.

[124] Y. Yang, X. Chen, and G. Gopalakrishnan, "Inspect: A runtime model checker for multithreaded C programs," Technical Report UUCS-08-004, University of Utah, Tech. Rep., 2008.

[125] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *PLDI '14*, 2014, pp. 337–348.

[126] J. Huang, "Stateless model checking concurrent programs with maximal causality reduction," in *PLDI '15*, 2015, pp. 165–174.

[127] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, "Assertion guided symbolic execution of multithreaded programs," in *ESEC/FSE 2015*, 2015, pp. 854–865.

[128] S. Guo, M. Kusano, and C. Wang, "Conc-iSE: incremental symbolic execution of concurrent software," in *ASE 2016*, 2016, pp. 531–542.

[129] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *ISSTA 2018*. New York, NY, USA: ACM, 2018, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213874

[130] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities," in *CCS '17*. New York, NY, USA: ACM, 2017, pp. 2155–2168. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134073

[131] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart Greybox Fuzzing," *CoRR*, vol. abs/1811.09447, pp. 1–16, 2018.

[132] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *SP '19*, vol. 00, 2019, pp. 279–293. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2019.00017

[133] C. Liu, D. Zou, P. Luo, B. B. Zhu, and H. Jin, "A Heuristic Framework to Detect Concurrency Vulnerabilities," in *ACSAC 2018*, 2018, pp. 529–541.

[134] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: identifying vulnerable code for vulnerability assessment through program metrics," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 60–71. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00024

[135] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 533–544. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338975

[136] A. R. Bernat and B. P. Miller, "Anywhere, Any-time Binary Instrumentation," in *PASTE '11*. New York, NY, USA: ACM, 2011, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/2024569.2024572

[137] A. Helin. (2018) radamsa - a general-purpose fuzzer. [Online]. Available: https://gitlab.com/akihe/radamsa

[138] H. Wang, X. Xie, S.-W. Lin, Y. Lin, Y. Li, S. Qin, Y. Liu, and T. Liu, "Locating vulnerabilities in binaries via memory layout recovering," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 718–728. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338966

[139] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *PLDI 2011*, 2011, pp. 283–294.

[140] N. V. Database. (2019) CVE-2019-9169 Detail. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-9169

[141] I. FIRST.org. (2019) Common Vulnerability Scoring System v3.0: Specification Document. [Online]. Available: https://www.first.org/cvss/specification-document

[142] S. R. Peter Mell, Karen Scarfone. (2019) A Complete Guide to the Common Vulnerability Scoring System Version 2.0. [Online]. Available: https://www.first.org/cvss/v2/guide

[143] A. P. Fuchs, A. Chaudhuri, and J. Foster, "SCanDroid : Automated Security Certification of Android Applications," University of Maryland, Tech. Rep. CS-TR-4991, November 2009.

[144] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014.

[145] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *CCS '14*. New York, NY, USA: ACM, 2014, pp. 1329–1341.

[146] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *ICSE '15*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 280–291.

[147] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 408–419. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180222

[148] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *Journal of Functional Programming*, vol. 15, no. 2, pp. 131–177, Mar. 2005.

[149] J. Landauer and T. Redmond, "A lattice of information," in *6th IEEE Computer Security Foundations Workshop - CSFW'93, Franconia, New Hampshire, USA, June 15-17, 1993, Proceedings*. IEEE Computer Society, 1993, pp. 65–70.

[150] Android, "Requesting permissions at run time." [Online]. Available: https://developer.android.com/training/permissions/requesting.html

[151] A. Developers, "Binder," https://developer.android.com/reference/android/os/Binder.html, 2017, online, accessed on 07-July-2017.

[152] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *MobiSys '11*, New York, NY, USA, 2011, pp. 239–252.

[153] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2-3, pp. 167–187, Jan. 1996.

[154] A. Developers, "Permissionchecker — android developers," https://developer.android.com/reference/android/support/v4/content/PermissionChecker.html, 2017, online, accessed on 07-July-2017.

[155] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Sep. 2003.

[156] H. Chen, A. Tiu, Z. Xu, and Y. Liu, "A permission-dependent type system for secure information flow analysis," *CoRR*, vol. abs/1709.09623, 2017. [Online]. Available: http://arxiv.org/abs/1709.09623

[157] T. M. , R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional verification and refinement of concurrent value-dependent noninterference," in *CSF '16*, June 2016, pp. 417–431.

[158] N. Polikarpova, J. Yang, S. Itzhaky, and A. Solar-Lezama, "Type-driven repair for information flow security," *CoRR*, vol. abs/1607.03445, 2016. [Online]. Available: http://arxiv.org/abs/1607.03445

[159] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "A dependent security type system for concurrent imperative programs," *Archive of Formal Proofs*, Jun. 2016, http://isa-afp.org/entries/Dependent_SIFUM_Type_Systems.shtml, Formal proof development.

[160] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional security-preserving refinement for concurrent imperative programs," *Archive of Formal Proofs*, Jun. 2016, http://isa-afp.org/entries/Dependent_SIFUM_Refinement.shtml, Formal proof development.

[161] T. Murray, "Short Paper: On High-Assurance Information-Flow-Secure Programming Languages," in *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, 2015.

[162] X. Li, F. Nielson, and H. Riis Nielson, "Future-dependent Flow Policies with Prophetic Variables," in *the 2016 ACM Workshop*. New York, NY, USA: ACM Press, 2016, pp. 29–42.

[163] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *the Twentieth International Conference*. New York, New York, USA: ACM Press, 2015, pp. 503–516.

[164] X. Li, F. Nielson, H. R. Nielson, and X. Feng, "Disjunctive Information Flow for Communicating Processes." *TGC*, 2015.

[165] L. Lourenço and L. Caires, "Dependent information flow types," in *POPL '15*, New York, NY, USA, 2015, pp. 317–328.

[166] L. Lourenço and L. Caires, "Information flow analysis for valued-indexed data security compartments," in *Lecture Notes in Computer Science*, vol. 8358. Springer, 2014, pp. 180–198.

[167] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," *Journal of Functional Programming*, vol. 23, no. 4, pp. 402–451, 2013.

[168] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," in *POPL 2012*. ACM, 2012, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103669

[169] H. Mantel, D. Sands, and H. Sudbrock, "Assumptions and guarantees for compositional noninterference," in *CSF 2011*. IEEE Computer Society, 2011, pp. 218–232. [Online]. Available: https://doi.org/10.1109/CSF.2011.22

[170] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *SP '11*. IEEE Computer Society, 2011, pp. 165–179.

[171] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in fine," in *Programming Languages and Systems, 19th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 6012. Springer, 2010, pp. 529–549.

[172] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control." *International Journal of Information Security*, vol. 6, no. 2-3, pp. 67–84, 2007.

[173] S. Tse and S. Zdancewic, "Run-time principals in information-flow type systems," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, 2007.

[174] P. Li and D. Zhang, "Towards a flow- and path-sensitive information flow analysis," in *CSF '17*, Aug 2017, pp. 53–67.

[175] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL '06*. New York, NY, USA: ACM, 2006, pp. 79–90.

[176] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC Enforcement on Android," *USENIX Security Symposium*, 2016.

[177] S. Lortz, H. Mantel, A. Starostin, T. Bahr, D. Schneider, and A. Weber, "Cassandra: Towards a certifying app store for android," in *SPSM '14*, New York, NY, USA, 2014, pp. 93–104.

[178] H. Gunadi, "Formal certification of non-interferent android bytecode (dex bytecode)," in *ICECCS '15*, Washington, DC, USA, 2015, pp. 202–205.

[179] A. Chaudhuri, "Language-based security on android," in *PLAS '09*, New York, NY, USA, 2009, pp. 1–7.

[180] G. C. Necula and P. Lee, "Proof-carrying code," School of Computer Science, Carnegie Mellon University, Tech. Rep., 1996, cMU-CS-96-165.

[181] S. Lin, J. Sun, T. K. Nguyen, Y. Liu, and J. S. Dong, "Interpolation guided compositional verification (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 65–74.

[182] S.-W. Lin, Y. Liu, J. Sun, J. S. Dong, and É. André, "Automatic compositional verification of timed systems," in *FM 2012: Formal Methods*, D. Giannakopoulou and D. Méry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 272–276.

[183] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[184] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: ACM, 2016, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897856

[185] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving android malware and dynamic loading technique,"

IEEE Transactions on Information Forensics and Security, vol. 12, no. 7, pp. 1529–1544, July 2017.

[186] H. Chen. (2018) FOT the Fuzzer. [Online]. Available: https://sites.google.com/view/fot-the-fuzzer

[187] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program learning," *CoRR*, vol. abs/1807.05620, pp. 1–15, 2018. [Online]. Available: http://arxiv.org/abs/1807.05620

[188] S. Nagy and M. Hicks, "Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing," *CoRR*, vol. abs/1812.11875, pp. 1–16, 2018. [Online]. Available: http://arxiv.org/abs/1812.11875

[189] C. Lv, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimize Mutation Scheduling for Fuzzers," in *USENIX Security '19*, 2019, pp. 1949–1966.

[190] L. Zhao, Y. Duan, H. Yin, and J. X. W. University), "Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing," in *NDSS '19*. San Diego, CA USA: The Internet Society, 2019, pp. 1–15.

[191] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *CCS '18*. New York, NY, USA: ACM, 2018, pp. 2095–2108. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243849

[192] S. Guo, M. Wu, and C. Wang, "Adversarial Symbolic Execution for Detecting Concurrency-Related Cache Timing Leaks," *CoRR*, vol. abs/1807.03280, 2018. [Online]. Available: http://arxiv.org/abs/1807.03280

[193] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic Race Detection with LLVM Compiler - Compile-Time Instrumentation for ThreadSanitizer," in *RV 2011*, 2011, pp. 110–114.

[194] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels," in *SP 2018*, 2018, pp. 661–678. [Online]. Available: https://doi.org/10.1109/SP.2018.00017

[195] M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing Racy Tests," *SIGPLAN Not.*, vol. 50, no. 6, pp. 175–185, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737998

[196] K. Sen, "Effective Random Testing of Concurrent Programs," in *ASE '07*. New York, NY, USA: ACM, 2007, pp. 323–332. [Online]. Available: http://doi.acm.org.ezlibproxy1.ntu.edu.sg/10.1145/1321631.1321679

[197] Z. Lai, S. C. Cheung, and W. K. Chan, "Detecting Atomic-set Serializability Violations in Multithreaded Programs Through Active Randomized Testing," in *ICSE '10*. New York, NY, USA: ACM, 2010, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806836

[198] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT '08*, 2008, pp. 72–81.

[199] P. Joshi, M. Naik, K. Sen, and D. Gay, "An effective dynamic analysis for detecting generalized deadlocks," in *ESEC/FSE '10*, 2010, pp. 327–336.

[200] T. Bergan, D. Grossman, and L. Ceze, "Symbolic execution of multithreaded programs from arbitrary program contexts," in *OOPSLA 2014*, 2014, pp. 491–506.

[201] Y. Shoshitaishvili, "preeny," 2015. [Online]. Available: https://github.com/zardus/preeny

[202] C. Wang, M. Said, and A. Gupta, "Coverage Guided Systematic Concurrency Testing," in *ICSE '11*. New York, NY, USA: ACM, 2011, pp. 221–230. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985824

[203] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[204] I. S. LLC, "Convert, Edit, Or Compose Bitmap Images @ ImageMagick," 1990. [Online]. Available: https://www.imagemagick.org/script/index.php

[205] V. Terragni and M. Pezzè, "Effectiveness and Challenges in Generating Concurrent Tests for Thread-safe Classes," in *ASE 2018*. New York, NY, USA: ACM, 2018, pp. 64–75. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238224

[206] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," in *ASPLOS 2008*. New York, NY, USA: ACM, 2008, pp. 329–339. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346323

[207] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and Accurate Static Data-race Detection for Concurrent Programs," in *CAV'07*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 226–239. [Online]. Available: http://dl.acm.org/citation.cfm?id=1770351.1770386

[208] K. Apinis, H. Seidl, and V. Vojdani, "How to Combine Widening and Narrowing for Non-monotonic Systems of Equations," in *PLDI '13*. New York, NY, USA: ACM, 2013, pp. 377–386. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462190

[209] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in *ESEC/FSE '07*, 2007, pp. 205–214.

[210] P. Joshi, M. Naik, C. Park, and K. Sen, "CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs," in *CAV 2009*, 2009, pp. 675–681.

[211] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *ISCA 2009*, 2009, pp. 325–336.

[212] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic Shared-Memory Multiprocessing," *IEEE Micro*, vol. 30, no. 1, pp. 40–49, Jan 2010.

[213] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in *ESEC/FSE 2007*. ACM, 2007, pp. 205–214.

[214] M. Naik, A. Aiken, and J. Whaley, *Effective static race detection for Java*. ACM, 2006, vol. 41, no. 6.

[215] (2014) discussion about fair comparisons between fuzzers. [Online]. Available: https://groups.google.com/forum/#!topic/afl-users/aC4cEL1Wtv4

[216] (2014) Cyber Grand Challenge. [Online]. Available: https://github.com/CyberGrandChallenge/

[217] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *SP '10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 497–512.

[218] B. Wu, B. Zhang, S. M. Wen, M. J. Li, Q. Zhang, and C. J. Tang, "Directed Fuzzing Based on Dynamic Taint Analysis for Binary Software," in *Applied Mechanics and Materials*, vol. 571. Trans Tech Publications, 8 2014, pp. 539–545.

[219] T. Parr. (2018) ANTLR (ANother Tool for Language Recognition). [Online]. Available: http://www.antlr.org/

[220] Intel. (2018) Intel XED. [Online]. Available: https://intelxed.github.io/

[221] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing Seed Selection for Fuzzing," in *USENIX Security '14*, 2014, pp. 861–875. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[222] W. Wang, H. Sun, and Q. Zeng, "SeededFuzz: Selecting and Generating Seeds for Directed Fuzzing," in *TASE '16*, 2016, pp. 49–56.

[223] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS '13*. ACM Press, 2013, pp. 511–522.

[224] V. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *ASE '16'*. ACM Press, 2016, pp. 543–553.

[225] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[226] G. Fraser and A. Arcuri, "The Seed is Strong: Seeding Strategies in Search-Based Software Testing," in *ICST '12'*. IEEE Computer Society, 2012, pp. 121–130.

[227] G. Roelofs. libpng. [Online]. Available: http://www.libpng.org/pub/png/libpng.html

[228] SQLite. (2018) SQLite: a SQL database engine. [Online]. Available: https://www.sqlite.org/index.html

[229] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits," in *CCS '17*. New York, NY, USA: ACM Press, 2017, pp. 2139–2154.

[230] Google. (2017) fuzzer-test-suite, bignum. [Online]. Available: https://github.com/google/fuzzer-test-suite/tree/master/openssl-1.1.0c

[231] ClusterFuzz-External. (2017). [Online]. Available: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=407

[232] D. Evans, "The Internet of Things How the Next Evolution of the Internet Is Changing Everything," Cisco Internet Business Solutions Group, Tech. Rep., 2011.

[233] J. B. Crawford. (2018) A survey of some free fuzzing tools. [Online]. Available: https://lwn.net/Articles/744269/

[234] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Empirical Software Engineering and Verification," in *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, B. Meyer and M. Nordio, Eds. Berlin, Heidelberg: Springer-Verlag, 2012, ch. Search Based Software Engineering: Techniques, Taxonomy, Tutorial, pp. 1–59.

[235] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012.

[236] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[237] R. Guo, "MongoDB's JavaScript fuzzer," *Communications of the ACM*, vol. 60, no. 5, pp. 43–47, 2017.

[238] M. Harman and R. M. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.

[239] X. Wang, Y. Jhi, S. Zhu, and P. Liu, "STILL: Exploit Code Detection via Static Taint and Initialization Analyses," in *ACSAC '08*. IEEE Computer Society, 2008, pp. 289–298.

[240] B. Steensgaard, "Points-to Analysis in Almost Linear Time," in *POPL '96*. ACM Press, 1996, pp. 32–41.

[241] T. W. Reps, "Program Analysis via Graph Reachability," in *ILPS '97*. MIT Press, 1997, pp. 5–19.

[242] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C programs," in *PLDI '95*, 1995, pp. 1–12.

[243] *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: https://www.ndss-symposium.org/ndss2016/

[244] G. Developers. (2016) Request app permissions. [Online]. Available: https://developer.android.com/training/permissions/requesting

[245] Q. Sun, A. Banerjee, and D. A. Naumann, "Modular and constraint-based information flow inference for an object-oriented language," in *Static Analysis*, R. Giacobazzi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 84–99.

[246] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993.

[247] J. van Rest, D. Boonstra, M. Everts, M. van Rijn, and R. van Paassen, *Designing Privacy-by-Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 55–72.

[248] D. F. C. Brewer and M. J. Nash, "The chinese wall security policy," in *Proceedings. 1989 IEEE Symposium on Security and Privacy*, May 1989, pp. 206–214.

[249] Y. Takata and H. Seki, *Automatic Generation of History-Based Access Control from Information Flow Specification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 259–275.

[250] D. Liu, "Bytecode verification for enhanced jvm access control," in *ARES '07*, Washington, DC, USA, 2007, pp. 162–172.

[251] A. Banerjee and D. A. Naumann, "A simple semantics and static analysis for stack inspection," in *Festschrift for Dave Schmidt*. EPTCS, 2013, pp. 284–308.

[252] M. Pistoia, A. Banerjee, and D. A. Naumann, "Beyond stack inspection: A unified access-control and information-flow security model," in *SP '07*, Washington, DC, USA, 2007, pp. 149–163.

[253] T. F. Lunt, "Aggregation and inference: facts and fallacies," in *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, May 1989, pp. 102–109.

[254] M. Kennedy and R. Sulaiman, "Following the wi-fi breadcrumbs: Network based mobile application privacy threats," in *ICEEI 2015*, Aug 2015, pp. 265–270.

[255] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *ACSAC '12*, New York, NY, USA, 2012, pp. 51–60.

[256] A. Nanevski, A. Banerjee, and D. Garg, "Dependent type theory for verification of information flow and access control policies," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, no. 2, pp. 6:1–6:41, Jul. 2013.

[257] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 129–142, Oct. 1997.

[258] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers, "Sharing Mobile Code Securely with Information Flow Control," *IEEE Symposium on Security and Privacy*, pp. 191–205, 2012.

[259] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *POPL '99*, New York, NY, USA, 1999, pp. 228–241.

[260] G. Barthe, D. Pichardie, and T. Rezk, "A certified lightweight non-interference java bytecode verifier," in *Proceedings of the 16th European Symposium on Programming*, ser. ESOP'07, Berlin, Heidelberg, 2007, pp. 125–140.

[261] Z. Fang, W. Han, and Y. Li, "Permission based Android security: Issues and countermeasures," *Computers & Security*, vol. 43, no. C, pp. 205–218, Jun. 2014.

[262] C. Marforio, A. Francillon, and S. Capkun, "Application collusion attack on the permission-based security model and its implications for modern smartphone systems," 2011.

[263] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *ICSE '15*, Piscataway, NJ, USA, 2015, pp. 280–291.

[264] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *CCS '11*. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[265] G. Barthe and T. Rezk, "Non-interference for a JVM-like language." *TLDI*, pp. 103–112, 2005.

[266] G. Barthe, T. Rezk, and D. A. Naumann, "Deriving an Information Flow Checker and Certifying Compiler for Java." *IEEE Symposium on Security and Privacy*, pp. 230–242, 2006.

[267] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, Oct. 2009.

[268] W. J. Bowman and A. Ahmed, "Noninterference for free," in *ICFP 2015*. New York, NY, USA: ACM, 2015, pp. 101–113.

[269] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering inconsistent security policy enforcement in the android framework," in *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[270] Android, "Manifest.permission." [Online]. Available: https://developer.android.com/reference/android/Manifest.permission.html

[271] E. Duan, "Dresscode and its potential impact for enterprise, september 2016." [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/dresscode-potential-impact-enterprises/

[272] M. Kumar, "Beware! new android malware infected 2 million google play store users." [Online]. Available: http://thehackernews.com/2017/04/android-malware-playstore.html