Binary-level directed fuzzing for complex vulnerabilities

Manh-Dung Nguyen

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Manh-Dung NGUYEN

Thèse dirigée par **Roland GROZ**, Professeur
et codirigée par **Sebastien BARDIN**, CEA Paris-Saclay
et **Matthieu LEMERRE**

préparée au sein du **Laboratoire Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Guidage du test en frelatage de codes binaires pour la détection de vulnérabilités logicielles complexes

Binary-level directed fuzzing for complex vulnerabilities

Thèse soutenue publiquement le **30 mars 2021**,
devant le jury composé de :

Monsieur AURELIEN FRANCILLON
PROFESSEUR ASSOCIE, EURECOM, Rapporteur
Monsieur JACQUES KLEIN
PROFESSEUR, Université du Luxembourg, Rapporteur
Madame MARIE-LAURE POTET
PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES, Examinatrice
Madame VALERIE VIET TRIEM TONG
PROFESSEUR, ESE CESSON, Examinatrice
Monsieur ROLAND GROZ
PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES, Directeur de thèse

To my respectable grandfather, my first ever teacher and my longtime friend Van-Liem NGUYEN

Acknowledgements

First, I would like to thank my advisors, Prof. Roland Groz (Université Grenoble Alpes), Dr. Sébastien Bardin & Dr. Matthieu Lemerre (CEA LIST), and Dr. Richard Bonichon (Tweag I/O), for their constant encouragement and guidance through my doctoral work. Thank you for giving me this opportunity to pursue my research interests on fuzzing and your kind support in many aspects of my PhD journey in France. I really appreciate your supervision and have learned much from you, such as how to conduct a research project, formal writing styles and Ocaml programming language.

Second, I feel greatly privileged to have met, discussed and learnt from many talented colleagues at CEA LIST/LSL in the BINSEC group, especially Dr. Benjamin Farinier, Frédéric Recoules, Mathilde Ollivier, Yaëlle Vinçont, Lesly-Ann Daniel, Olivier Nicole, Grégoire Menguy, Guillaume Girol, Dr. Michaël Marcozzi, Soline Ducousso and Charles B Mamidisetti. I would like to thank Dr. Josselin Feist for his insightful discussions, his thesis and his static analyzer that inspired me to follow this research direction. Also, I had a great time discussing fuzzing research and collaborating with Yaëlle, Prof. Christophe Hauser and Wei-Cheng Wu.

Third, this doctoral work could not be complete without the evaluation of my thesis examiners, Prof. Aurélien Francillon, Prof. Jacques Klein, Prof. Marie-Laure Potet and Prof. Valérie Viet Triem Tong. I would like to thank them for their time for serving on the committees of my thesis defend, the careful reading and the constructive comments. I would also like to thank Prof. Olivier Levillain for the mid-term evaluation and his insightful suggestions on my thesis proposals.

Fourth, I would like to thank my supervisors, my collaborators and my friends at National University of Singapore where I started my research career: Prof. Joxan Jaffar, Prof. Abhik Roychoudury, Dr. Duc-Hiep Chu, Prof. Van-Thuan Pham, Prof. Marcel Böhme, Prof. Sergey Mechtaev, Dr. Yannic Noller, Dr. Naipeng Dong, Dr. Quang-Trung Ta and Duc-Than Nguyen, just to name a few. I also feel lucky to be able to work with great fuzzing researchers such as Marcel and Thuan.

Last, but not least, I want to express my gratitude to my beloved family who has been an endless source of love, motivation and encouragement for me: my grandfather Van-Liem, my parents Manh-Quan and Hong-Tham, my younger brother Manh-Quang, my wife Hoai-Thuong, my son Nhat-Minh (Bean thoi) and also many other great friends in France. I dedicate this doctoral work to my grandfather Van-Liem and my son Nhat-Minh.

Abstract

Fuzzing is a popular security testing technique consisting in generating massive amount of random inputs, very effective in triggering bugs in real-world programs. Although recent research made a lot of progress in solving fuzzing problems such as magic numbers and highly structured inputs, detecting complex vulnerabilities is still hard for current feedback-driven fuzzers, even in cases where the targets are known (*directed fuzzing*). In this thesis, we consider the problem of guiding fuzzing to detect complex vulnerabilities such as Use-After-Free (UAF), as bug-triggering paths must satisfy specific properties of those bug classes. UAF is currently identified as one of the most critical exploitable vulnerabilities and has serious consequences such as data corruption and information leaks. Firstly, we provide a detailed survey on Directed Greybox Fuzzing, which is the core technique of this thesis, aiming to perform stress testing on predefined targets like recent code changes or vulnerable functions. Secondly, we propose new directed fuzzing techniques tailored to detecting UAF vulnerabilities in binary code that we have proven effective and efficient in both bug reproduction and patch testing. Thirdly, we show that our directed techniques can be fruitfully generalized to other typestate bugs like buffer overflows. Finally, our proposed techniques have been implemented in the open-source tools BINSEC/UAFUZZ and BINSEC/TYPEFUZZ, helping to find security vulnerabilities in real-world programs (39 new bugs, 17 CVEs were assigned and 30 bugs were fixed).

Keywords: Automated vulnerability detection, Greybox fuzzing, Directed fuzzing, Bug reproduction, Patch testing, Use-After-Free.

Résumé

Le test en frelatage (*fuzz testing* ou *fuzzing*) est une technique de test de sécurité populaire consistant à générer une quantité massive d'entrées aléatoires, très efficace pour déclencher des bogues dans des programmes du monde réel. Bien que des recherches récentes aient permi beaucoup de progrès dans la résolution de problèmes de fuzzing tels que les nombres magiques et les entrées hautement structurées, la détection de vulnérabilités complexes est toujours difficile pour les fuzzers actuels, même si les cibles sont connues (*fuzzing dirigé*). Dans cette thèse, nous considérons le problème du guidage du fuzzing pour détecter des vulnérabilités complexes telles que Use-After-Free (UAF), car les chemins de déclenchement de tels bogues demandent de satisfaire des propriétés très spécifiques. Le bogue UAF est actuellement identifié comme l'une des vulnérabilités exploitables les plus critiques et a des conséquences graves telles que la corruption des données et les fuites d'informations. Tout d'abord, nous fournissons une étude détaillée sur le Directed Greybox Fuzzing, qui est la technique de base de cette thèse, visant à effectuer des tests de résistance sur des cibles prédéfinies comme les changements récents ou les fonctions vulnérables. Deuxièmement, nous proposons de nouvelles techniques de fuzzing dirigées adaptées à la détection des vulnérabilités UAF au niveau du binaire que nous avons prouvées efficaces et efficientes à la fois pour la reproduction de bogues et le test de correctifs. Troisièmement, nous montrons que nos techniques dirigées peuvent être généralisées avec succès à d'autres bogues qui violent les propriétés comme les débordements de tampon. Enfin, les techniques que nous avons proposées ont été implémentées dans les outils open-source BINSEC/UAFUZZ and BINSEC/TYPEFUZZ, aidant à trouver des vulnérabilités de sécurité dans les programmes du monde réel (39 nouveaux bogues, 17 CVEs ont été attribués et 30 bogues ont été corrigés).

Mots-clés: Détection automatisée des vulnérabilités, Test en frelatage, Fuzzing dirigé, Reproduction de bogues, Test de correctifs, Use-After-Free.

Contents

Acknowledgements v

Abstract vii

Résumé ix

Contents xii

List of Figures xiv

Listings xv

List of Tables xvii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Challenges and Objectives . . . 4
- 1.3 Contributions . . . 8
  - 1.3.1 Scientific contributions . . . 8
  - 1.3.2 Technical contributions . . . 9
  - 1.3.3 Publications and talks . . . 10
- 1.4 Outline . . . 11

2 Background 13
- 2.1 Memory Corruption Vulnerabilities . . . 13
- 2.2 Automated Vulnerability Detection . . . 15
  - 2.2.1 Dynamic Symbolic Execution . . . 16
  - 2.2.2 Search-based Software Testing . . . 16
  - 2.2.3 Coverage-guided Greybox Fuzzing . . . 17
  - 2.2.4 Hybrid Fuzzing . . . 20
- 2.3 Conclusion . . . 21

3 A Survey of Directed Greybox Fuzzing 23
- 3.1 Introduction . . . 23
  - 3.1.1 Formalization of the Directed Fuzzing Problem . . . 24

3.1.2 Applications of Directed Fuzzing . . . 25
3.1.3 Differences between Directed and Coverage-based Fuzzing . . . 26
3.2 Overview . . . 26
3.2.1 Workflow . . . 26
3.2.2 Core Algorithm . . . 27
3.3 Input Metrics . . . 28
3.3.1 Distance metric . . . 28
3.3.2 Covered function similarity metric . . . 30
3.4 Differences between Source- and Binary-based Directed Fuzzing . . . 30
3.5 Limitations & Future Directions . . . 31
3.6 Conclusion . . . 32

4 Binary-level Directed Fuzzing for Use-Afer-Free Vulnerabilities 33
4.1 Introduction . . . 34
4.2 Motivation . . . 35
4.3 The UAFuzz Approach . . . 37
4.3.1 Bug Trace Flattening . . . 38
4.3.2 Seed Selection based on Target Similarity . . . 40
4.3.3 UAF-based Distance . . . 42
4.3.4 Power Schedule . . . 44
4.3.5 Postprocess and Bug Triage . . . 46
4.4 Experimental Evaluation . . . 47
4.4.1 Research Questions . . . 47
4.4.2 Evaluation Setup . . . 47
4.4.3 UAF Bug-reproducing Ability (RQ1) . . . 49
4.4.4 UAF Overhead (RQ2) . . . 53
4.4.5 UAF Triage (RQ3) . . . 56
4.4.6 Individual Contribution (RQ4) . . . 57
4.4.7 Patch Testing & Zero-days . . . 58
4.4.8 Threats to Validity . . . 61
4.5 Related Work . . . 62
4.5.1 Directed Greybox Fuzzing . . . 62
4.5.2 Coverage-based Greybox Fuzzing . . . 62
4.5.3 UAF Detection . . . 62
4.5.4 UAF Fuzzing Benchmark . . . 63
4.6 Conclusion . . . 64

5 Implementation 65
5.1 Introduction . . . 65
5.2 Preprocessing . . . 67
5.2.1 Bug trace generation . . . 67
5.2.2 BinIda Plugin . . . 68
5.3 Core Fuzzing Engine . . . 70

5.3.1 Debugging with afl-showmap . . . 70
5.3.2 Overhead . . . 70
5.4 Examples . . . 71
5.4.1 Application 1: Bug Reproduction . . . 71
5.4.2 Application 2: Patch Testing . . . 75
5.5 Conclusion . . . 76

6 Typestate-guided Directed Fuzzing **79**
6.1 Introduction . . . 79
6.2 The TypeFuzz Approach . . . 81
6.2.1 Different Bug Characteristics . . . 81
6.2.2 Adapted Techniques . . . 82
6.3 Evaluation . . . 83
6.3.1 Research Questions . . . 83
6.3.2 Evaluation Setup . . . 84
6.3.3 Bug-reproducing Ability (RQ1) . . . 85
6.3.4 Crash Triage (RQ2) . . . 86
6.3.5 Target Reaching (RQ3) . . . 87
6.4 Patch Testing . . . 89
6.5 Conclusion . . . 89

7 Conclusion **91**
7.1 Summary . . . 91
7.1.1 Research problems . . . 91
7.1.2 Scientific contributions . . . 92
7.1.3 Technical contributions . . . 93
7.2 Perspectives . . . 93

Acronyms **97**

Bibliography **99**

List of Figures

1.1 Number of UAF bugs in NVD [nvd20] 5

2.1 Typestate for Use-After-Free and Double-Free bugs. 14
2.2 Different approaches of exploring the input space, where • are selected inputs to be mutated, × are generated inputs and shaded area are interesting space [BAS+19]. 15

3.1 Workflow of Directed Greybox Fuzzing (DGF) (different components compared to Coverage-guided Greybox Fuzzing (CGF) are in gray). 27
3.2 Difference between node distance defined in terms of arithmetic mean versus harmonic mean. Node distance is shown in the white circles. The targets are marked in gray [BPNR17]. 29

4.1 Overview of UAFuzz. 37
4.2 Bug trace of CVE-2018-20623 (UAF) produced by Valgrind. 38
4.3 Reconstructed Dynamic Calling Tree (DCT) from CVE-2018-20623 (bug trace from Figure 4.2). The preorder traversal of this tree is simply $0 \rightarrow 1 \rightarrow 2 \rightarrow 3(n_{alloc}) \rightarrow 4(n_{free}) \rightarrow 5 \rightarrow 6(n_{use})$. 39
4.4 Precision lattice for Target Similarity Metrics. 41
4.5 Example of a call graph. Favored edges are in red. 43
4.6 Summary of fuzzing performance (RQ1). 49
4.7 TTE in seconds of 4 fuzzers except for subjects marked with "(m)" for which the unit is minute (lower is better). AQ, AG, HK and UF denote AFL-QEMU, AFLGoB, HawkeyeB and UAFuzz, respectively. 51
4.8 Fuzzing queue of UAFuzz for `yasm-issue-91`. Selected inputs to be mutated are highlighted in brown. Potential inputs are in the horizontal dashed line. 52
4.9 Summary of fuzzing performance of 4 fuzzers against our benchmark, except `CVE-2017-10686` due to compilation issues of AFLGo. 52
4.10 Global overhead (RQ2). 54
4.11 Average instrumentation time in seconds (except `CVE-2017-10686` due to compilation issues of AFLGo). 54
4.12 Average instrumentation time in seconds. 54
4.13 Total executions done in all runs. 55
4.14 Average triaging time in seconds. 56

4.15 Summary of bugs triage (RQ3). 57
4.16 Impact of each components (RQ4). 58
4.17 The bug trace of CVE-2018-6952 (Double Free) produced by Valgrind. . 61

5.1 Overview of UAFuzz workflow. 66
5.2 Code structure of UAFuzz. 66
5.3 DCT of the program mjs generated by our preprocessing script. 67
5.4 Formats of the files extracted using IDA Pro. 68
5.5 A basic block of `mjs_mkstr()` in the program mjs. 69
5.6 UAFuzz shared memory – extended layout (x86-64) 71
5.7 The identified (non-) cut edges of this example given the bug trace. C, N denotes cut and non-cut, respectively. 72
5.8 DCT of this example. 73
5.9 The stack traces of this example produced by Valgrind. 73
5.10 Call graph and important CFGs (only show the first instruction of each basic block) of this example produced by the BinIda plugin. 74
5.11 The user interface of UAFuzz. 75

6.1 Overview of TypeFuzz. 80
6.2 The stack trace of CVE-2016-4488 produced by Valgrind. 81
6.3 DCT of CVE-2016-4488. 82
6.4 Summary of fuzzing performance (RQ1). 86

Listings

2.1 Code snippet illustrating a UAF bug. 14
3.1 Call pattern 1. 30
3.2 Call pattern 2. 30
4.1 Motivating example. 36
4.2 Code fragment of GNU patch pertaining to the UAF vulnerability `CVE-2018-6952`. 60
5.1 Outputs of afl_showmap. 70
5.2 A simple example. 73
5.3 UAFuzz's output when fuzzing the latest version of GNU Patch with the timeout 6 hours. 75
6.1 The expected bug trace of CVE-2016-4488. 82

List of Tables

2.1 Overview of recent greybox fuzzers classified by technique and publication date. S, B and S&B represent respectively source code, binary and both levels of analysis (based on benchmarks). ①: complex structure problem, ②: code coverage problem, ③: complex bugs, ④: directedness problem, ⑤: human interaction, ⑥: parallel fuzzing, ⑦: anti-anti-fuzzing and ⑧: seed generation. . . . 18

4.1 Summary of existing greybox fuzzing techniques. . . . 34
4.2 Overview of main techniques of greybox fuzzers. Our own implementations are marked with *. . . . 47
4.3 Overview of our evaluation benchmark. . . . 48
4.4 Summary of bug reproduction of UAFuzz compared to other fuzzers against our fuzzing benchmark. Statistically significant results $\hat{A}_{12} \geq 0.71$ are marked as bold. . . . 49
4.5 Bug reproduction on 4 fuzzers against our benchmark. Statistically significant results $\hat{A}_{12} \geq 0.71$ are marked as bold. *Factor* measures the performance gain as the μTTE of other fuzzers divided by the μTTE of UAFuzz. 50
4.6 Bug reproduction of AFLGo against our benchmark except `CVE-2017-10686` due to compilation issues of AFLGo. Numbers in red are the best μTTEs. . . . 53
4.7 Average number of triaging inputs of 4 fuzzers against our tested subjects. For UAFuzz, the TIR values are in parentheses. . . . 56
4.8 Bug reproduction on 4 fuzzers against our benchmark. $\hat{A}_{12A}$ and $\hat{A}_{12U}$ denote the Vargha-Delaney values of AFLGoB and UAFuzz. Statistically significant results for $\hat{A}_{12}$ (e.g., $\hat{A}_{12A} \leq 0.29$ or $\hat{A}_{12U} \geq 0.71$) are in bold. Numbers in red are the best μTTEs. . . . 58
4.9 Summary of zero-day vulnerabilities reported by our fuzzer UAFuzz (**32** new bugs including **13** new UAF bugs, **10** CVEs were assigned and **23** bugs were fixed). . . . 59
4.10 Summary of existing benchmarks. . . . 63
4.11 Detailed view of our evaluation benchmark. . . . 64

5.1 Detailed results of BinIda in processing our evaluation benchmark in Table 4.3. . . . 69

6.1 Overview of our evaluation benchmark. 85
6.2 Summary of bug reproduction of TypeFuzz compared to other fuzzers against our fuzzing benchmark. Statistically significant results $\hat{A}_{12} \geq 0.71$ are marked as bold. 86
6.3 Average number of correct crashing inputs of 4 fuzzers against our tested subjects. Numbers in red are the best values. 87
6.4 Average TTR of 4 fuzzers against our tested subjects, given *only one target basic block*. Numbers in red are the best μTTRs. 88
6.5 Average TTR in seconds of 4 fuzzers against our tested subjects, given *a full bug trace*. Numbers in red are the best μTTRs. The difference values of 3 directed fuzzers compared to Table 6.4 are in parentheses. 88
6.6 Summary of zero-day vulnerabilities reported by our fuzzer TypeFuzz. HBO, NPD denote heap buffer overflow and NULL pointer dereference, respectively. 89

Chapter 1

Introduction

Contents

1.1 Context 1
1.2 Challenges and Objectives 4
1.3 Contributions 8
1.3.1 Scientific contributions 8
1.3.2 Technical contributions 9
1.3.3 Publications and talks 10
1.4 Outline 11

> "*If you know the enemy and know yourself, you need not fear the result of a hundred battles ...*"
>
> — Sun Tzu, *The Art of War*

1.1 Context

Context In the era of automation technologies, software controls every aspect of our life, from daily needs to a big human dream of exploring the universe like NASA's Mars 2020 Perseverance Rover [nas20]. However, programs are written by human beings and therefore they contain bugs, which can in turn become security vulnerabilities. A simple bug like the one that has negative impacts on user experience (e.g., a wrong user interface display) can be harmless from a security perspective. More severely, a logical vulnerability can cause a program crash (e.g., denial-of-service attacks) or can be exploitable, allowing attackers to inject and execute malicious code to obtain high privileges. In this case, these vulnerabilities cause serious damages from significant financial losses to even people's deaths. For example, Heartbleed (CVE-2014-0160) [hea20] – a very well-known critical vulnerability in the popular OpenSSL cryptographic software library – caused by an implementation defect can leak secret keys and compromise the integrity of communications of web services. On

April 2019, Israel failed to land an unmanned spacecraft on the moon's surface due to a software bug with its engine system [ber20]. Even worse, due to the same reason, four crew members died and two were injured in an air force cargo plane that crashed on a test flight in Spain [air20]. The accident could be more catastrophic if the similar vulnerability exists in a civil aircraft. Furthermore, there is still much controversy on the ethical issues of automated decision-making Artificial Intelligence (AI) systems like self-driving cars [eth20] when put into use in reality.

All the examples above raise a question: *How can we avoid the serious consequences brought by software defects?* One answer is simple and obvious, by testing software programs and testing them in a very careful and systematic manner.

Software testing plays an important role in multiple phases through the software development life cycle, from high-level design testing to low-level source code testing and also in the maintenance after the software is released. Concretely, software security testing aims to generate test cases that show the vulnerability, a.k.a., Proof-of-Concept (PoC), if it actually exists. Once developers have more clues on the bugs, they can debug the buggy software to find the root cause and eventually fix them. Indeed, Google and Facebook have paid $6.5 million [goo20] and $2.2 million [fac20] to external security researchers who discovered and submitted bugs in their products in 2019 alone respectively. Moreover, hackers have earned $100 million in bug bounties on the number one hacker-powered security platform HackerOne [hac20].

Existing automated vulnerability analysis Finding bugs early is indeed crucial in the vulnerability management process. Security experts usually perform a manual code audit or employ automated approaches with the help of more powerful computing resources to hunt vulnerabilities. With the growth of the complexity of software systems, manual testing becomes much more challenging, tedious and time-consuming. In contrast, automated testing has been widely used and have common techniques as follows.

Static analysis approaches [CKK+12, BBC+10, cod20] perform the analysis without executing the Program Under Test (PUT). Although these approaches have shown their effectiveness in proving the presence of program bugs, only potential buggy locations are provided to the developers. Apart from high false positive rate, another common weakness of all static detectors is therefore their inability to infer concrete test cases triggering the bugs. Consequently, some extra efforts are still needed for developers to investigate and verify whether reports produced by static tools are actually real bugs.

Formal methods, such as abstract interpretation [CC77], which are common techniques used by software engineers to design safety-critical systems, are historically not designed to find bugs. Hence, similar to static testing approaches, formal methods produce possible false positives and cannot generate a concrete PoC (only model-based buggy program traces). Furthermore, the scalability issues limit the practical usefulness of formal method techniques on large programs. More recent approaches, like bounded model checking [KT14] and Symbolic Execution (SE) (e.g., KLEE [CDE+08], S2E [CKC11] and SymCC [PF20]), are able to find bug-triggering inputs, but still suffer from scalability

issues.

Dynamic testing approaches, such as fuzz testing (a.k.a., fuzzing) [MFS90,pea20,afl20i], run the PUT and generate inputs as witnesses for program bugs, making the debugging phase easier and less error-prone. While those dynamic approaches can indeed find PoCs, they are either not automated (e.g., standard testing), require manual intervention (e.g., for property checking [CH11]), or remain at a too shallow blackbox level, such as random testing [Ham02]. Yet, recent so-called *greybox fuzzing* [afl20a] methods can find PoCs, trigger deep bugs and work on binary code.

Greybox fuzzing Fuzzing [MFS90], which was first introduced by Miller et al. in 1990 to test the reliability of UNIX tools is a simple yet very effective testing technique for automated detection of vulnerabilities. On one hand, from an input generation perspective (e.g., by using mutation operators or input models), fuzzing techniques can be classified as *mutation-* or *generation-based fuzzing*. On the other hand, fuzzing techniques can be categorized in three (3) parts depending on the degree of program analysis: *blackbox fuzzing*, *greybox fuzzing* and *whitebox fuzzing*. The *blackbox fuzzing* simply considers the PUT as a black box, thus this technique does not require any program analysis but rather mutates inputs blindly. In contrast to the blackbox fuzzing, the *whitebox fuzzing* mainly employs heavy-weight program analysis such as SE to systematically discover as many feasible paths as possible of the PUT. However, the whitebox fuzzing has a scalability issue due to the well-known path explosion problem.

The recent rise of *greybox fuzzing* [MHH+19, BCR21] in both academia and industry, such as Springfield [Mic20] and OneFuzz [one20] of Microsoft, AFL [afl20i], OSS-Fuzz [oss20a] and ClusterFuzz [clu20] of Google, shows its ability to find a large number of bugs in real-world applications [afl20d]. The greybox fuzzing, which is placed in between the blackbox fuzzing and the whitebox fuzzing, employs light-weight program analysis and uses feedback information to effectively guide the fuzzers at runtime. Technically, Coverage-guided Greybox Fuzzing (CGF), such as AFL [afl20i] and libFuzzer [lib20a], leverages code coverage information in order to guide input generation toward new parts of the PUT, exploring as many program states as possible in the hope of triggering crashes. For example, libFuzzer is able to trigger the Heartbleed vulnerability within several seconds [lib20b].

Directed greybox fuzzing In some cases where the vulnerable code is known (e.g., from bug reports or in dangerous functions like string copy operations or relevant buggy code on other platforms), an ideal fuzzer should spend its time budget on quickly reaching target locations without wasting efforts exploring unrelated or well-tested code. To address this limitation, the concept of Directed Greybox Fuzzing (DGF) [BPNR17,CXL+18,WZ20] was introduced in 2017. For instance, if OpenSSL's developers performed directed fuzzing as soon as the commit was submitted to the code base, then the Heartbleed vulnerability would have been found as it was introduced. While the main goal of coverage-guided fuzzing is to cover as many program states as possible in a limited time, directed

fuzzing aims to perform stress testing on pre-selected potentially vulnerable target locations. DGF has therefore many applications to different security contexts: (1) *bug reproduction* [JO12, PNRR15, BPNR17, CXL+18], (2) *patch testing* [MC13, PLL+19, BPNR17] or (3) *static analysis report verification* [CMW16, LZY+19]. Depending on the application, target locations are originated from bug stack traces, patches or static analysis reports.

Problems Despite tremendous progress in many aspects in the past few years (e.g., magic bytes comparison [laf20, LCC+17, LS18, ASB+19], deep execution [SGS+16, RJK+17, CC18], lack of directedness [BPNR17, CXL+18] and complex file formats [BAS+19, YWM+19, PBS+19, FDC19], etc.), current (directed or not) greybox fuzzers still have a hard time finding ***complex vulnerabilities***. For example, OSS-Fuzz [oss20a, oss20b] or recent greybox fuzzers [BPNR17, RJK+17, YWM+19] only found a small number of Use-After-Free (UAF). Moreover, in cases where the vulnerable events of a UAF bug are identified (e.g., from the bug report), existing directed fuzzers are too generic and lack of specific design strategies to effectively detect this type of bug.

Finding bugs is hard, finding complex vulnerabilities is even harder as bug-triggering paths may satisfy very *specific properties* of specific bug classes. Böhme [Böh19] had a vision about several types of complex bugs for current software testing techniques in general and fuzzing in particular: non-interference, flaky bugs, bugs outside the fuzzer's search space or due to extremely rare program behaviors. In this case, further analysis is required to better understand characteristics of the target bugs and adapt software testing techniques, especially random ones like (directed) fuzzing, to boost the directedness to meet complex bug-triggering conditions.

Scope In the scope of this thesis, we focus on *mutation-based (directed) greybox fuzzing techniques*, which are behind the success of many recent vulnerability detection tools. More specifically, we aim to tackle the issues of *directed fuzzing* discussed above by first investigating specific properties of "*hard-to-detect*" vulnerabilities and carefully tuning several of key components of directed fuzzing to the specifics of these bug classes.

My thesis was performed in collaboration with two laboratories: my hosting laboratory CEA LIST – the Safety and Security Lab of the *Commissariat à l'Énergie Atomique et aux energies alternatives* and the LIG (Laboratoire d'Informatique de Grenoble) of Université Grenoble Alpes in the VASCO team – *Validation de Systèmes, Composants et Objets logiciels*. My research work was supported by the H2020 project C4IIoT under the Grant Agreement No 833828 and the FUI project CAESAR.

1.2 Challenges and Objectives

Complex vulnerabilities Classic memory corruptions identified by Common Weakness Enumeration (CWE) like buffer overflows (CWE-121, CWE-122) [CPM+98, HSNB13], NULL pointer dereference (CWE-476) [HP07, FMRS12] or integer overflows (CWE-190) [WWLZ09, MLW09, DLRA15] have been well studied. In contrast, recent vulnera-

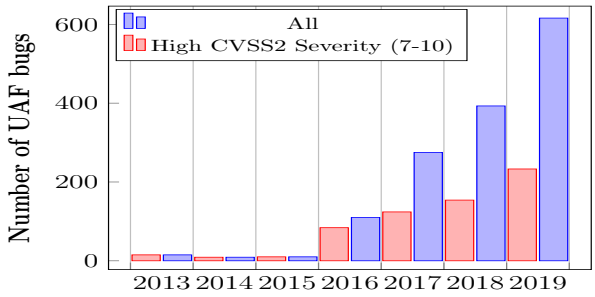


Figure 1.1: Number of UAF bugs in NVD [nvd20]

bility classes such as UAF (CWE-415, CWE-416) or type confusion have not received much attention in the literature.

Our insight is that several vulnerabilities can be considered as the violation of typestate properties [SY86]. Typestate properties can aid program understanding, define type systems [DF04] that prevent programmers from causing typestate errors or even derive static analysis [FGRY03, FYD+08] to verify whether a given program violates typestate properties, especially in formal verification. For example, the sequence of finite-state machine $\langle alloc \rightarrow free \rightarrow use \rangle$ is a witness of triggering the UAF bug. In other words, UAF bug-triggering paths in the program must satisfy the given typestate property. Hua et al. proposed Machine Learning (ML)-guided typestate analysis for static UAF detection by leveraging ML techniques to tackle the problem of high overhead of typestate analysis, making it scalable to real-world programs [YSCX17].

Indeed, there are recently more fuzzing work targeting uncommon, complex bug classes: performance bugs [PZKJ17, LPSS18], UAF [WXL+20], concurrency bugs [muz20], memory consumption bugs [WWL+20], hypervisor bugs [SAA+17] and Database Management System (DBMS) bugs [ZCH+20]. They share the same conclusion: *vulnerability-oriented greybox fuzzers have better fuzzing performance than general ones in detecting specific bug classes*. Those fuzzers bring insights for our research work and we aim to develop new fuzzing techniques to effectively detect typestate vulnerabilities.

In the same vein of existing vulnerability-oriented fuzzers, we focus on UAF bugs. They appear when a heap element is used after having been freed. Figure 1.1 shows that the numbers of UAF bugs has increased in the National Vulnerability Database (NVD) [nvd20]. According to the Project Zero team at Google, 63% of exploited 0-day vulnerabilities fall under memory corruption, with half of those memory corruption bugs being UAF vulnerabilities in 2019 [rev20]. They are currently identified as *one of the most critical exploitable vulnerabilities* due to the lack of mitigation techniques compared to other types of bugs such as buffer overflows. They may have serious consequences such as data corruption, information leaks and denial-of-service attacks. However, fuzzers targeting the detection of UAF bugs confront themselves with the following challenges.

- **Complexity** – Exercising UAF bugs require to generate inputs triggering a *sequence* of 3 events – *alloc*, *free* and *use* – *on the same memory location*, spanning multiple

functions of the PUT, whereas buffer overflows only require a single out-of-bound memory access. This combination of both *temporal* and *spatial* constraints is extremely difficult to meet in practice;

- **Silence** – UAF bugs often have *no observable effect*, such as segmentation faults. In this case, fuzzers simply observing crashing behaviors do not detect that a test case triggered such a memory bug. Sadly, popular profiling tools such as AddressSanitizer (ASan) [SBPV12] or VALGRIND [NS07] cannot be used in a fuzzing context. While ASan not only requires the source code but also involves high runtime overhead, there are still no practical lightweight approach to fuzz binaries with VALGRIND due to its heavyweight instrumentation.

Hypotheses We make the following assumptions in the remainder of this thesis:

- Tested binaries are typically compiled from C/C++ programs using a classic compiler to avoid obfuscated binaries;
- We assume that there exists stack traces of known vulnerabilities that we aim to reproduce, so that our fuzzers can take them as input to guide the dynamic strategies;
- The architecture of the tested binaries is the *Intel x86*, yet our techniques can be easily adapted to work on other architectures or even at source level;
- Anti-fuzzing techniques [GAAH19, JHS+19] are not considered in our work;
- For the sake of simplicity we suppose that all the transition systems that we study are deterministic, which implies that non-deterministic or flaky bugs are out of the scope of this thesis.

Directed greybox fuzzing Actually, current state-of-the-art directed fuzzers, namely AFLGO [BPNR17] and HAWKEYE [CXL+18], fail to address these challenges. First, they are too generic and therefore do not cope with the specificities of UAF such as temporality – their guidance metrics do not consider any notion of sequenceness. Second, they are completely blind to UAF bugs, requiring to send all the many generated seeds to a profiling tool for an expensive extra check. Finally, current implementations of source-based DGF fuzzers typically suffer from an expensive instrumentation step [afl20f], e.g., AFLGO spent nearly 2 hours compiling and instrumenting `cxxfilt` (Binutils). Our main goal is therefore to develop an effective directed fuzzing technique towards UAF vulnerabilities in different security contexts.

Bug reproduction. We focus mainly on reproducing bugs, which is the most common practical application of DGF [JO12, BPNR17, YZC+17, CXL+18, LZY+19]. It consists in generating PoC inputs of disclosed vulnerabilities given bug report information. It is especially needed since only 54.9% of usual bug reports can be reproduced due to missing information and users' privacy violation [MCY+18]. Even with a PoC provided in the bug

report, developers may still need to consider all corner cases of the bug in order to avoid regression bugs or incomplete fixes. In this situation, providing more bug-triggering inputs becomes important to facilitate and accelerate the repair process.

Patch testing. Another interesting use case is to check if an existing vulnerability is corrected in a more recent version. The main idea is to use bug stack traces of *known* UAF bugs to guide testing on the *patched* version of the PUT – instead of the buggy version as in bug reproduction. The benefit from the bug hunting point of view [gpz20] is both to try finding buggy or incomplete patches and to focus testing on *a priori* fragile parts of the code, possibly discovering bugs unrelated to the patch itself. For instance, an incomplete fix for a NULL pointer dereference CVE-2017-15023 led to a new bug of the same type CVE-2017-15939 in GNU Binutils 2.29 [cve20a].

Static reports verification. We are interested in investigating the effectiveness and efficiency of a hybrid testing technique combining static analysis and directed fuzzing to detect UAF bugs. While the two aforementioned applications may rely on information of disclosed bugs to address the low reproducibility problem and test relevant code, this application justifies reports produced by static analyzers and subsequently generates PoCs in case the bug actually exists with no prior information. In this setting, static reports allow to narrow the fuzzing search space and effectively focus the fuzzer's effort on potentially vulnerable components. However, the bug-finding performance of the hybrid approach may depend on the quality of static reports (e.g., including real buggy locations).

Binary-level analysis It is indeed more important to find bugs at binary level since the source code of some critical programs is not always available or relies on third-party libraries. Furthermore, two different compilers can produce two different binaries with different behaviors due to the undefined behaviors of the language. Thus, the ability of analyzing and testing software at binary level allows us to mitigate and take into account the expected interpretation of these undefined behaviors. It also brings more flexibility in selecting the PUT.

Fuzzing benchmarks Existing widely-used fuzzing benchmarks which contain either artificial common vulnerabilities [DGHK+16, RPDGH18, rod20, cgc20] or artificial programs [NIS20] raise a strong need of having a suitable benchmark for evaluation of complex vulnerability-oriented fuzzers. Actually, RODE0DAY [rod20], a continuous bug finding competition, recognizes that fuzzers should aim to cover new bug classes like UAF in the future [FLDGB19], moving further from the widely-used LAVA [DGHK+16] synthetic bug corpora which only contains buffer overflows. Furthermore, FUZZBENCH [fuz20], which is a free service that evaluates fuzzers at large scale on a wide variety of real-world benchmarks including Google Fuzzer Testsuite [gft20], currently supports only coverage-guided fuzzers.

Open-source projects & Zero-day vulnerabilities While automated vulnerability detection has been an active research area, the security community still lacks available solutions as some tools are still closed-source. By making our tools and benchmark available

as open-source projects, we hope to facilitate future fuzzing work in general and complex vulnerability-oriented (directed) fuzzing in particular. Furthermore, finding zero-day vulnerabilities in real-world programs shows that our proposed techniques works well in different security contexts. Finally, by reporting new bugs of open-source projects, it allows the developers to analyze and fix the bugs, especially critical vulnerabilities, as early as possible to make the software more robust.

1.3 Contributions

Overall, our contributions in this thesis are at several levels. For *science contributions*, we had three (3) main contributions, which are a survey of directed fuzzing and the design, implementation and evaluation of two directed fuzzers targeting complex vulnerabilities in binary code. For *technical contributions*, we released our new directed fuzzers, our UAF fuzzing benchmark and also contributed to the open-source binary analysis platform BINSEC [bin20]. To sum up, our contributions led to 3 research articles, 4 talks and 39 new bugs with 17 CVEs.

1.3.1 Scientific contributions

A survey of directed fuzzing Chapter 3 introduces a detailed survey on DGF focusing on its security applications, formal definitions, challenges, existing solutions, current limitations and promising future directions. We discuss in details directed fuzzing techniques proposed in the state-of-the-art to provide a better understanding on the core techniques of DGF behind our contributions in this thesis.

Directed fuzzing for UAF vulnerabilities We design the first directed greybox fuzzing technique *tailored to UAF bugs* in Chapter 4. Especially, we systematically revisit the three (3) main ingredients of directed fuzzing including selection heuristic, power schedule and input metrics and specialize them to UAF. It is worth noting that we aim to find an input fulfilling both control-flow (temporal) and runtime address (spatial) conditions to trigger the UAF bug. We solve this problem by bringing UAF characteristics into DGF in order to generate more potential inputs reaching targets in sequence w.r.t. the UAF expected bug trace.

- We propose three (3) dynamic seed metrics specialized for UAF vulnerabilities detection. The *distance metric* approximates how close a seed is to all target locations, and takes into account the need for the seed execution trace to cover the three UAF events in order. The *cut-edge coverage metric* measures the ability of a seed to take the correct decision at important decision nodes. Finally, the *target similarity metric* concretely assesses how many targets a seed execution trace covers at runtime;

- Our seed selection strategy favors seeds covering more targets at runtime. The power scheduler determines the energy for each selected seed based on its metric scores

during the fuzzing process;

- Finally, we take advantage of our previous metrics to pre-identify likely-PoC inputs that are sent to a profiling tool (here VALGRIND) for bug confirmation, avoiding many useless checks.

Directed fuzzing for typestate vulnerabilities As we start with a bug trace that is actually a sequence of method calls in bug reproduction, the ordering of target locations is indeed important. Overall, similar to the directed fuzzer UAFUZZ, TYPEFUZZ is made out of several components including seed selection, power schedule and crash triage. It is worth noting that different bugs have different characteristics in terms of bug traces and runtime behaviors. Thus, we adapt the *ordering-based input metrics* initially tailored to UAF bugs to find other widespread typestate vulnerabilities, such as buffer overflow or NULL pointer dereference, in a more general context, in Chapter 6.

Evaluation on practical applications To evaluate the effectiveness of the proposed techniques, we compare our fuzzers UAFUZZ and TYPEFUZZ with state-of-the-art coverage-guided and directed greybox fuzzers against the popular fuzzing benchmarks and also known bugs of real-world security-critical programs.

Bug reproduction. Our evaluation demonstrates that our fuzzers are highly effective and significantly outperform state-of-the-art competitors. In addition, our fuzzers enjoy both low instrumentation and runtime overheads. Furthermore, we also show that improvements of each key ingredient of UAFUZZ are proven complementary and contribute to the final fuzzing performance in finding UAF vulnerabilities.

Patch testing. Our fuzzers are also proven effective in patch-oriented testing, leading to the discovery of 39 unknown bugs (17 CVEs) in widely-used projects like GNU Binutils, GPAC, MuPDF and GNU Patch (including 4 buggy patches). So far, 30 bugs have been fixed. Interestingly, by using the stack trace of the Double-Free CVE-2018-6952, UAFUZZ successfully discovered an incomplete bug fix CVE-2019-20633 [dfp20] in the same bug class in the latest version of GNU Patch with a slight difference of the bug stack trace.

1.3.2 Technical contributions

Open-source toolchains We develop open-source toolchains on top of the state-of-the-art greybox fuzzer AFL [afl20a] and the binary analysis platform BINSEC [bin20], named UAFUZZ [uaf20b] in Chapters 4 and 5 and TYPEFUZZ in Chapter 6, implementing the above method for directed fuzzing over binary codes and enjoying small overhead. We have implemented a BINSEC plugin computing *statically* distance and cut-edge information, consequently used in the instrumentation of our fuzzers – note that Call Graph (CG) and Control Flow Graph (CFG) are retrieved from the IDA PRO [ida20] binary database. On the *dynamic* side, we have modified AFL-QEMU to track covered targets, dynamically compute seed scores and power functions. Finally, a small script automates bug triaging.

UAF fuzzing benchmark We construct and openly release a fuzzing benchmark dedicated to UAF [uaf20a], comprising 30 real bugs from 17 widely-used projects (including the few previous UAF bugs found by existing directed fuzzers), in the hope of facilitating future UAF fuzzing evaluation.

1.3.3 Publications and talks

Our contributions above led to the writing of the following research outputs in security conferences:

- **Manh-Dung Nguyen**, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre, "*Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities*", The 23nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20), 2020.

- **Manh-Dung Nguyen**, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre, "*About Directed Fuzzing and Use-After-Free: How to Find Complex & Silent Bugs?*", Black Hat USA, 2020.

This thesis was presented in the PhD Student Symposium of several security workshops in French as follows:

- **Manh-Dung Nguyen**, "*Directed Fuzzing for Use-After-Free Vulnerabilities Detection*", Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI '20 – Session doctorants), 2020.

- **Manh-Dung Nguyen**, "*Directed Fuzzing for Use-After-Free Vulnerabilities Detection*", 19èmes Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL '20 – Session doctorants), 2020.

We currently submitted the following journal article that presents all major contributions of this thesis.

- **Manh-Dung Nguyen**, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre, "*Binary-level Directed Fuzzing for Complex Vulnerabilities*", *under submission* to IEEE Transactions on Software Engineering (TSE), 2021.

Before my PhD, I contributed to the seminal work on directing greybox fuzzing, which is the core technique discussed systematically in this thesis.

- Marcel Böhme, Van-Thuan Pham, **Manh-Dung Nguyen**, Abhik Roychoudhury, "*Directed Greybox Fuzzing*", Conference Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '17), 2017.

1.4 Outline

This thesis is split into seven (7) chapters as follows:

- Chapter 1 first introduces the overview of my thesis including context, challenges, objectives and contributions;
- Chapter 2 presents software vulnerabilities, especially memory corruption bugs and then provides some background about greybox fuzzing in general;
- Chapter 3 introduces our first principle contribution by providing a comprehensive survey on DGF, focusing on its formal definitions, challenges, existing solutions, current limitations and promising future directions;
- Chapter 4 introduces our second principle contribution by proposing UAFuzz, which is the first binary-level directed fuzzer to detect UAF bugs with detailed evaluations of two practical applications: bug reproduction and patch testing;
- Chapter 5 discusses the implementation of our fuzzer UAFuzz and show its usage via examples;
- Chapter 6 introduces our third principle contribution by showing the generality of our directed fuzzing techniques to detect typestate bugs in Chapter 6 with some detailed experiments;
- Chapter 7 concludes this thesis with a discussion on our research problems, our contributions and possible future extensions of this work.

Chapter 2

Background

Contents

2.1 Memory Corruption Vulnerabilities **13**
2.2 Automated Vulnerability Detection **15**
2.2.1 Dynamic Symbolic Execution 16
2.2.2 Search-based Software Testing 16
2.2.3 Coverage-guided Greybox Fuzzing 17
2.2.4 Hybrid Fuzzing 20
2.3 Conclusion **21**

This chapter presents a background on software bugs, especially memory related ones and common existing automated vulnerability detection techniques. Here we focus on dynamic testing techniques, especially Coverage-guided Greybox Fuzzing (CGF), that can provide concrete bug-triggering inputs to help developers understand the root cause of the bugs and fix them.

2.1 Memory Corruption Vulnerabilities

Memory safety violations [SLR+19] which are among the most severe security vulnerabilities in C/C++ programs have been studied extensively in the literature. These vulnerabilities cause programs to crash, allowing their exploitation to lead to serious consequences such as information leakage, code injection, control-flow hijacking and privilege escalation.

Spatial safety violations, such as buffer overflows, happen when dereferencing a pointer out of the bounds of its intended pointed object. *Temporal safety violations* occur when dereferencing a pointer to an object which is no longer valid (i.e., the pointer is *dangling*). Observing a good stack discipline is usually easy and suffices to avoid bugs involving pointer to stack objects. Thus, the most serious temporal memory violation involve pointers to objects allocated on the heap; those are called Use-After-Free (UAF) bugs.

Furthermore, typestate analysis [SY86, FYD+08] represents another approach for detecting statically temporal memory safety violations. The typestates of an object are

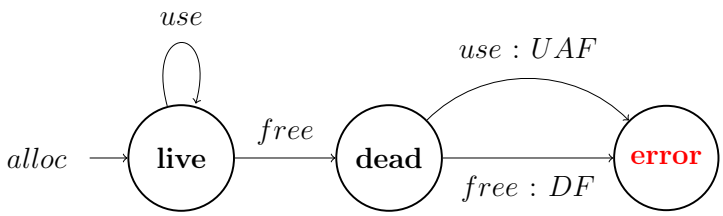


Figure 2.1: Typestate for Use-After-Free and Double-Free bugs.

tracked by statically analyzing all the statements that affect the state transitions along all the feasible paths in the program [YSCX17]. *Typestate bugs* often indicate violations to certain safety conditions or security properties. Common memory corruption vulnerabilities can be seen as typestate bugs. For example, $\langle nullify \rightarrow dereference \rangle$ is a witness for triggering a NULL pointer dereference. Similarly, $\langle alloc \rightarrow free \rightarrow use \rangle$ is the sequence of states violating typestate property of UAF bugs, as illustrated in Figure 2.1. In other words, a UAF warning for an object o is reported when a free call `free(p)` reaches a use call `use(q)`, which denotes a memory access on the same object pointed by q, along a control-flow path, where $*p$ and $*q$ are aliases (i.e., p and q point to o).

Use-After-Free *Use-After-Free* (UAF) bugs happen when dereferencing a pointer to a heap-allocated object which is no longer valid (i.e., the pointer is *dangling*), as shown in Listing 2.1. Note that Double-Free (DF) is a special case, where the dangling pointer is used to call `free()` again.

```
1 char *buf = (char *) malloc(BUF_SIZE);
2 ...
3 free(buf); // pointer buf becomes dangling
4 ...
5 strncpy(buf, argv[1], BUF_SIZE-1); // Use-After-Free
```

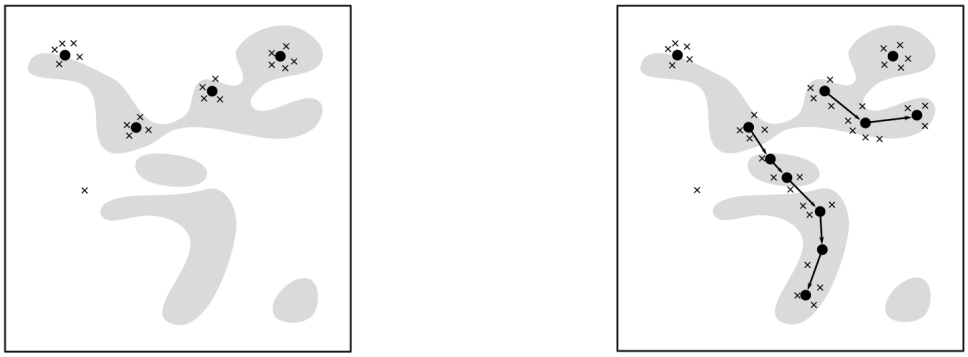
Listing 2.1: Code snippet illustrating a UAF bug.

UAF-triggering conditions Triggering a UAF bug requires to find an input whose execution covers in sequence *three UAF events*: an allocation (*alloc*), a *free* and a *use* (typically, a dereference), violating typestate property in Figure 2.1, *all three referring to the same memory object*. Furthermore, this last *use* generally does not make the execution immediately crash, as a memory violation crashes a process only when it accesses an address outside of the address space of the process, which is unlikely with a dangling pointer. Thus, UAF bugs go often unnoticed and are a good vector of exploitation [You15, LSJ+15]. For instance, attackers can overwrite a return address when the dangling pointer is an escaped pointer to a local variable and points to the stack [SPWS13].

2.2 Automated Vulnerability Detection

Existing automated vulnerability detection techniques can be considered as search over the input space of the Program Under Test (PUT) to identify bug-triggering inputs. As the input space of real-world programs is usually large, we aim to wisely search for *interesting inputs* that trigger new program behavior (e.g., new code lines or new paths). The intuition behind the search is that the more new states we explore, the more error states or bugs we can find.

Definition 1. The goal of automated vulnerability detection is to find states $s_0 \in \mathbb{S}_0$ such that $s_0 \rightarrow^* \Omega$, where $\mathbb{S}_0$ is a set of initial states, Ω denotes the *error state* and $\rightarrow$ describes the transitions between states.



(a) In a random manner (b) With some guidance

Figure 2.2: Different approaches of exploring the input space, where • are selected inputs to be mutated, × are generated inputs and shaded area are interesting space [BAS+19].

We focus on dynamic approaches. We can classify automated testing techniques depending on the manner they explore the input space. *Random techniques*, like blackbox fuzzing, explore the input space blindly. They mostly generate inputs near the initial seeds and are therefore unable to explore new interesting space or new program features, as shown in Figure 2.2a. To address this problem, *semi-random techniques*, like Search-based Software Testing (SBST) or CGF, still involve randomness but also provide feedback or dynamic guidance to decide which inputs are interesting to be kept in the extended corpus. The criteria of selecting interesting inputs (e.g., scoring function in SBST or code coverage in CGF) depend on the final goal of the testing phase. This strategy allows to gradually explore the new input space that is far from the initial corpus and consequently discover new paths of the PUT, as illustrated in Figure 2.2b.

However, due to randomness, these approaches shown in Figure 2.2 cannot make any guarantees that the PUT is free of errors after the testing process. *Systematic approaches*, like Symbolic Execution (SE), is even though costly but also more powerful as they can systematically explore the input space by targeting specific program paths and generating inputs covering these paths with the help of Satisfiability Modulo Theories (SMT) solvers.

Theoretically, given enough time, this strategy provides deterministic guarantees as being able to explore all feasible paths in the program. However, it is less practical, especially for real-world complex programs, due to the scalability issues and current limitations of SMT solvers.

We will introduce dynamic automated vulnerability detection techniques and their notable work: Dynamic Symbolic Execution (DSE) (§2.2.1), SBST (§2.2.2), CGF (§2.2.3) and hybrid methods particularly between CGF and other techniques (§2.2.4).

2.2.1 Dynamic Symbolic Execution

Similar to fuzzing, DSE or "*concolic*" execution [CS13] aims to generate new interesting inputs from existing corpus, but in a systematic way. First, it runs the program with the initial input and collects the *path constraints* on the input representing the execution path. For instance, assume we have an input variable x, we execute the program with the symbolic value $x = \alpha$ like normal SE and also the concrete value $x = 1$. Then, it negates one of those path constraints to represent an alternate path. Finally it employs a SMT constraint solver like Z3 [DMB08] to produce a satisfying input for the new path. Recently, there are more hybrid techniques that combine the efficiency of CGF and the effectiveness of DSE. Stephens et al. [SGS+16] designed Driller – the first hybrid fuzzing framework to overcome fuzzing roadblocks such as magic-bytes comparisons. QSYM [YLX+18] tackled the performance bottleneck of existing concolic executors by tightly integrating the symbolic emulation with the native execution using dynamic binary translation, making it possible to scale to find bugs in real-world software.

2.2.2 Search-based Software Testing

The key idea of SBST [McM04, McM11] is to employ local search algorithms for generating test data. Some real-world SBST tools for unit test generation are AUSTIN [LHG13] and EvoSuite [FA11]. Naturally, the concept of SBST is suitable in the fuzzing context. For instance, Szekeres et al. [Sze17] proposed Search-based Fuzzing (SBF) combining the scalability of fuzzers and the directionality of symbolic execution via several stochastic local search strategies directly on the target to find coverage-increasing inputs. [JC19] leverages Machine Learning (ML) to create a useful fitness function in the context of searching for executions satisfying a specific property particularly crash reproduction using fuzzing. We discuss the following key components of SBST, allowing to apply a search-based optimization technique in software testing.

Fitness function The principal role of the fitness function is to guide the search toward promising areas of the search space by scoring candidate solutions. The fitness function is problem-specific as it evaluates different points in the search space with respect to their *interestingness* for a specific goal. One of the most prominent examples of a fitness function is to reach and cover a target branch [WBS01]. In the fuzzing context [afl20a], a fitness function can evaluate the code coverage (e.g., lines, branches or functions are covered

by an input) at runtime to identify promising inputs to be mutated during the fuzzing loop. Furthermore, VUzzer [RJK+17] evaluates the fitness of an input depending on the interestingness of its execution paths (e.g., the number of covered non-error-handling basic blocks).

Search strategies Common search algorithms of SBST [Bac96] are Hill Climbing, Genetic Algorithms, or Simulated Annealing. Existing greybox fuzzers also use these algorithms to design their power schedules which determine how much "*energy*" (or fuzzing time) is assigned to a given input during fuzzing [BPR16].

Hill Climbing starts at a random input and evaluates its neighboring inputs in the search space using the fitness function. If a better candidate is found, Hill Climbing jumps to this new input and continues this process, until the neighborhood of the current input offers no better solutions (a.k.a., *local optimum*). In case where the found local optimum is not the *global optimum*, the strategy restarts the search again from a new initial position in the search space. AFLSmart [PBS+19] implements a Hill Climbing power schedule that assigns more energy to inputs with a higher degree of validity regarding the input format.

Simulated Annealing is similar to Hill Climbing, except that the movement around the search space is more stochastic. Particularly, it also takes random moves towards inputs with lower fitness scores (or worse solutions) with some probability to avoid getting stuck in local minimum. This algorithm uses the *temperature* parameter to regulate the acceptance of worse solutions. AFLGo [BPNR17] designs a Simulated Annealing-based power schedule that assigns more energy to a seed that is closer to the targets.

Genetic Algorithms are slightly different from the aforementioned local search strategies. While *local search algorithms* move from one point in the search space and always keep track of only one best solution, *Genetic Algorithms* maintain multiple solutions at the same time. Actually, state-of-the-art greybox fuzzers like AFL [afl20a] or libFuzzer [lib20a], employ Genetic Algorithms to increase code coverage.

2.2.3 Coverage-guided Greybox Fuzzing

While original approaches were completely blackbox and more or less akin to random testing, recent CGF [afl20a, lib20a] leverages *lightweight* program analysis to guide the search – typically through coverage-based feedback. As the name suggests, CGF is geared toward covering code in the large, in the hope of finding unknown vulnerabilities. Table 2.1 which classifies recent researches on CGF based on their techniques is representative rather than exhaustive (a more complete version could be found in the Systemization of Knowledge (SOK) paper [MHH+19]).

Code coverage Fuzzers aim to execute and observe the behavior of the programs with a huge number of inputs. To improve the performance of coverage-guided fuzzers, the primary goal is on getting this feedback as fast as possible at runtime. There are two common methods by which fuzzers can obtain the code coverage information. First, for

Table 2.1: Overview of recent greybox fuzzers classified by technique and publication date. S, B and S&B represent respectively source code, binary and both levels of analysis (based on benchmarks). ①: complex structure problem, ②: code coverage problem, ③: complex bugs, ④: directedness problem, ⑤: human interaction, ⑥: parallel fuzzing, ⑦: anti-anti-fuzzing and ⑧: seed generation.

Name	Article	Core techniques	Level of analysis	Open source	Direction
AFL [afl20a]	2015	State-of-the-art CGF	S&B	✓	②
LIBFUZZER [lib20a]	2017	State-of-the-art CGF	S	✓	②
AFLFast [BPR16]	CCS'16	AFL + power schedule	S	✓	②
SlowFuzz [PZKJ17]	CCS'17	LIBFUZZER + new mutation strategies	S	✓	③
PerfFuzz [LPSS18]	ISSTA'18	AFL + new mutation strategies	S	✓	③
FairFuzz [LS18]	ASE'18	AFL + new mutation strategies	S	✓	②
MOpt [LJZ+19]	UseSec'19	AFL + new mutation mechanisms	S	✓	②
TortoiseFuzz [WJL+]	NDSS'19	AFL + new coverage measurements	S	✓	②⑦
Driller [SGS+16]	NDSS'16	AFL ↔ SE	B	✓	②
Munch [OHPP18]	SAC'18	AFL → SE & SE → AFL	S	✓	②
T-Fuzz [PSP18]	S&P'18	AFL + program transformation + SE	B	✓	②
QSym [YLX+18]	UseSec'18	Hybrid fuzzing & DSE for binaries	B	✓	②
DigFuzz [ZDYX19]	NDSS'19	Hybrid fuzzing & DSE	S	✗	②
Pangolin [HYW+20]	S&P'20	AFL + Qsym + polyhedral path abstraction	S	✗	②
VUzzer [RJK+17]	NDSS'17	AFL-like + DTA + SA	B	✓	①②
Angora [CXL+18]	S&P'18	AFL-like + DTA + gradient descent	S	✓	②
TIFF [JRGB18]	ACSAC'18	VUzzer + type-inference	B	✗	①
Matryoshka [CLC19]	CCS'19	Angora + DTA for deeply nested branches	S	✗	②
GreyOne [GZC+20]	UseSec'19	Data flow sensitive fuzzing via DTA	S	✗	②
Steelix [LCC+17]	FSE'17	AFL + SA + instrumented comparison	B	✗	①
AFLGo [BPNR17]	CCS'17	AFL + SA + power schedule	S	✓	④
CollAFL [GZQ+18]	S&P'18	AFL + SA	S	✗	②
Hawkeye [CXL+18]	CCS'18	AFL-like + SA + power schedule	S	✗	④
Nezha [PTS+17]	S&P'17	LIBFUZZER + SA + differential testing	S	✓	②
ParmeSan [ÖRBG20]	UseSec'20	LIBFUZZER +	S	✓	④
Augmented-AFL [RBS17]	arxiv	AFL + several neural models	S	✗	②
Learn&Fuzz [GPS17]	ASE'17	Seq2seq model	S	✗	①
Neuzz [SPE+18]	SP'19	novel feed-forward Neural Networks	S	✓	②
EnFuzz [CJM+19]	UseSec'19	Ensemble diverse fuzzers	B	✓	②
FuzzGuard [ZLW+]	UseSec'20	AFLGO+ predict unreachable inputs	S	✗	④
HaCRS [SWD+17]	CCS'17	Human-in-the-loop for binaries	B	✗	⑤
IJON [ASAH]	S&P'20	AFL + an annotation mechanism	S	✓	⑤
perf-fuzz [XKMK17]	CCS'17	LIBFUZZER+ 3 new operating primitives	S	✓	⑥
PAFL [LJC+18]	FSE'18	AFL + a new parallel mechanism	S	✗	⑥
Skyfire [WCWL17]	S&P'17	Data-driven highly-structured seeds generation	S	✓	⑧

the static instrumentation, the compiler adds special code (e.g., an unique random number for AFL [afl20j]) at the start of each basic block to store the coverage information. This method is therefore fast and widely used in popular coverage-guided fuzzers like AFL, LIBFUZZER, etc. Second, in case where the source code is not available, fuzzers employ Dynamic Binary Instrumentation (DBI) to obtain such coverage information. For example, VUzzer and Steelix use PIN-based instrumentation [LCM+05], while Driller and T-Fuzz

rely on the QEMU-based instrumentation. Specifically, AFL supports various dynamic instrumentation mechanisms for DBI, such as QEMU, PIN [afl20g], DynamoRIO [afl20b], and Dyninst [afl20c]. However, this method suffers from the runtime overhead issues, consequently is slower than the static instrumentation method.

AFL Here, we discuss in details the state-of-the-art AFL [afl20j], which led to a significant amount of research on coverage-guided fuzzers.

- *Edge coverage.* Existing CGF mostly relies on the edge coverage. To track this coverage, AFL [afl20a] associates to each basic block a unique random *ID* during instrumentation. The coverage of the PUT on an input is collected as a set of pairs *(edge ID, edge hits)*, where *edge ID* of an edge $A \rightarrow B$ is computed as $ID_{A \rightarrow B} \triangleq (ID_A >> 1) \oplus ID_B$. Practically, *edge hits* values are bucketized to small powers of 2 (e.g., 1, 2, 3, 4-7, 8-15, 16-31, 32-127, and 128-255 times).

- *Seed prioritization.* A *seed* (input) is *favored* (selected) when it reaches under-explored parts of the code, and such favored seeds are then *mutated* to create new seeds for the code to be executed on.

- *Power schedule.* At the start of a new cycle, each input in the fuzzing queue is assigned an energy (a.k.a., fuzzing budget), which determines how many times each input is to be modified and executed. Particularly, AFL's power schedule employs several fitness heuristics depending on inputs' characteristics (e.g., input size, execution time with respect to the average or discovery time). For example, AFL doubles the assigned energy of an input exercising a new path.

Directions for fuzzing research We can distinguish eight (8) kinds of directions for improving fuzzing performance as follows:

1. **Complex structure** The randomness behind fuzzing means that it has a low probability of finding a solution to hard code such as magic byte comparisons or parsing, which usually depend on the input.

2. **Code coverage** In direct relation to the previous problem, fuzzing sometimes explores no more than the surface of the program or cannot explore deep paths in the PUT which are likely to trigger more interesting bugs.

3. **Complex bugs** Although CGF shows their ability to find various types of bugs (e.g., buffer overflows), complex bugs [Böh19] like UAF or non-deterministic bugs are still a big challenge for existing fuzzers.

4. **Directedness** CGF lacks the ability to drive the execution towards user-specified targets in the PUT – something useful for various testing scenarios such as patch testing or bug reproduction.

5. **Human interaction** Even though fuzzing is easy to setup and run, it is still hard for normal users or even developers to be actively involved in the fuzzing process (e.g., providing some dynamic guidance in case the fuzzers get stuck or a better visualization of fuzzing progress).

6. **Parallel fuzzing** Existing greybox fuzzers [afl20a, lib20a] support fuzzing in parallel to take advantage of powerful hardware resources. However, the proposed master-slave mechanism simply runs multiple instances and synchronizes coverage-increasing inputs. Consequently, this strategy is less efficient due to overlapped work between multiple fuzzing instances.

7. **Anti-anti-fuzzing** Recent anti-fuzzing techniques, such as Antifuzz [GAAH19] and Fuzzification [JHS+19], are proposed to hinder the fuzzing process from adversaries as much as possible. There is a very few work considering those advanced techniques in the fuzzing context. In other words, existing fuzzers may perform worse when fuzzing protected binaries.

8. **Seed generation** Seed generation is crucial for the efficiency of fuzzing, especially for highly-structured input format as random inputs produced by CGF are usually unable to pass the semantic checking. We aim to tackle the problem of generating a high quality test suite to improve the fuzzing performance.

2.2.4 Hybrid Fuzzing

To address the fuzzing research problems, existing work has improved internal components of greybox fuzzers. For example, AFLFast [BPR16] favors test cases covering rarely taken paths of the PUT, then introduces a power schedule to determine the time required to fuzz selected test cases. FairFuzz [LS18] introduces a mutation masking technique and changes test case selection strategy to increase code coverage. Lyu et al. [LJZ+19] proposes MOpt, a novel mutation scheduling scheme using Particle Swarm Optimization (PSO) algorithm, allowing mutation-based fuzzers to hunt bugs more efficiently. Moreover, [XKMK17] designs and implements three fuzzer-agnostic operating primitives to solve fuzzing performance bottlenecks and improve its scalability and performance on multi-core machines. Furthermore, existing work shows the effectiveness and efficiency of combining CGF with the following techniques.

Hybrid static analysis & fuzzing Static analysis can be used to gain general information about the PUT before running it. For example, Steelix [LCC+17] uses static analysis and a modified instrumentation to find magic bytes and mutate test cases according to comparisons present in the program. CollAFL [GZQ+18] improves greybox fuzzers' coverage accuracy with new hash algorithms for blocks.

Hybrid DSE & fuzzing DSE can be used to generate new test cases or afterwards to check crashes. Driller [SGS+16] does the former by using SE and a SMT solver like Z3 [DMB08] to generate test cases leading to new parts of the program when the fuzzer gets stuck. T-Fuzz [PSP18], on the other hand, applies semantic-preserving transformation to the program – which leads to false positives – then reproduces crashes on the original PUT using SE.

Hybrid dynamic taint analysis & fuzzing Dynamic Taint Analysis (DTA) can be used to gain information at runtime, especially about the execution of a given test case. For example, VUzzer [RJK+17] employs DTA to extract control and data flow features from the PUT to guide input generation. Angora [CC18] uses byte-level taint tracking and gradient descent to track unexplored branches and solve path constraints. Matryoshka [CLC19] employs taint analysis that allows fuzzers to explore deeply nested conditional statements.

Hybrid machine learning & fuzzing Recent research work explores how ML has been applied to address principal challenges in fuzzing for vulnerability detection [WJLL19, SRDK19]. Learn&Fuzz [GPS17] proposes a Recurrent Neural Network approach to automatically generate complex structured inputs like pdf files and increase the code coverage. NEUZZ [SPE+18] employs a dynamic neural program embedding to smoothly approximate a PUT's branch behavior.

Hybrid human-in-the-loop fuzzing Recently, the human-in-the-loop approach gained the attention of the fuzzing community. For example, Shoshitaishvili et al. [SWD+17] introduces the system HaCRS that allows humans to interact with the target application by analyzing the target and providing a list of strings relevant to the PUT's behavior. IJON [ASAH] leverages source-based annotations from a human analyst to guide the fuzzer to overcome roadblocks. Additionally, VisFuzz [ZWL+19] proposes an interactive tool for better understanding and intervening fuzzing process via runtime visualization.

2.3 Conclusion

In this chapter, we introduce common memory corruption bugs, especially typestate vulnerabilities such as UAF. Different dynamic testing techniques have been proposed in related work and each technique has its own pros and cons in terms of finding memory corruption bugs. Finally, we provide an overview about CGF with the state-of-the-art AFL, which is behind hundreds of high-impact vulnerability discoveries of real-world projects.

Chapter 3

A Survey of Directed Greybox Fuzzing

Contents

3.1 Introduction **23**
3.1.1 Formalization of the Directed Fuzzing Problem 24
3.1.2 Applications of Directed Fuzzing 25
3.1.3 Differences between Directed and Coverage-based Fuzzing . . . 26
3.2 Overview **26**
3.2.1 Workflow 26
3.2.2 Core Algorithm 27
3.3 Input Metrics **28**
3.3.1 Distance metric 28
3.3.2 Covered function similarity metric 30
3.4 Differences between Source- and Binary-based Directed Fuzzing **30**
3.5 Limitations & Future Directions **31**
3.6 Conclusion **32**

This chapter aims to introduce a detailed survey on Directed Greybox Fuzzing (DGF) focusing on its security applications, formal definitions, challenges, existing solutions, current limitations and promising future directions. This chapter is indeed important to provide a better understanding of the core techniques of DGF behind our contributions in this thesis.

3.1 Introduction

As previously discussed in §2.2.3, there are several research directions aiming to boost fuzzing performance of greybox fuzzing. One interesting direction is DGF [BPNR17, CXL+18] which aims at reaching a *pre-identified potentially buggy part of the code* from a *target* (e.g., patch, static analysis report), as often and fast as possible, since existing greybox fuzzers cannot be effectively directed. In particular, directed fuzzers follow the general principles and architecture of Coverage-guided Greybox Fuzzing (CGF), but adapt

the key components to their goal, essentially favoring seeds *"closer"* to the target rather than seeds discovering new parts of code.

DGF is indeed important to guide the search towards vulnerable code to reduce the fuzzing time budget and wisely use the hardware infrastructures for both developers and attackers. From the developers' point of view, they want to perform stress testing on new components instead of spending time to test well-tested or bug-free components again. From the attackers' point of view, starting with a recent bug fix or a list of potentially vulnerable functions as attack vectors gives them more chance to find bugs quickly in the target applications. Furthermore, according to Shin et al. [SW13], only 3% files of the large code base in Mozilla Firefox are buggy, thus covering all code paths of the large software is impossible and ineffective in practice.

3.1.1 Formalization of the Directed Fuzzing Problem

Definition 2 (Transition system)**.** A *transition system* is a tuple $\langle \mathbb{S}_0, \mathbb{S}, \rightarrow \rangle$, where $\mathbb{S}_0$ is a set of initial states, $\mathbb{S}$ is the set of all states, and $\rightarrow \in \mathbb{S} \times \mathbb{S}$ describes the admissible transitions between states.

As mentioned in the hypotheses in Chapter 1, we assume that all the transition systems that we study are deterministic, which implies that non-deterministic or flaky bugs are out of the scope of this thesis. A *deterministic transition system* is a transition system where the transition $\rightarrow$ is right-unique (i.e., the successor state is completely determined by the predecessor state, and $\rightarrow$ is a *partial function*).

Definition 3 (Complete state trace)**.** A *complete state trace* is a sequence $\langle s_0, \ldots, s_n \rangle$ of states (i.e., a pair of method calls and a memory state) such that

$$\forall i : 0 \leq i < n \Rightarrow s_i \rightarrow s_{i+1}$$

Assuming determinacy, the execution depends on an *input*, which is the set of all the values fed to the program. An *instruction trace* of an input is the sequence of program locations (e.g., addresses or lines of code) in a complete state trace. An *execution trace* of an input is the complete sequence of states executed by the program on this input. Clearly, the final goal of automated vulnerability detection in general and (directed) fuzzing in particular is to find an input whose execution trace ends with a visible error (e.g., a crash). Furthermore, by inspection of the state of a process when it crashes, we can extract a *stack trace*, which is the sequence of call stacks in a complete state trace.

Definition 4 (Reachability)**.** A *reachable* state s is a state such that

$$\exists s_0 \in \mathbb{S}_0 : s_0 \rightarrow^* s$$

Definition 5 (Matching input)**.** We say that s_0 *matches* a target instruction trace t or callstack trace T (e.g., written $s_0 \vDash t$) if the execution starting from s_0 passes through all the program locations in t or callstacks in T, respectively.

Automated vulnerability detection can be considered as a *search problem* over the input space to satisfy a specific condition. While existing directed symbolic execution approaches cast the reachability problem as iterative constraint satisfaction problem [CDE+08, MC13], as the state-of-the-art DGF, AFLGo [BPNR17] casts the reachability of target locations as an optimization problem and adopts a meta-heuristic to prioritize potential inputs. Depending on the application, a *target*, which is originated from bug stack traces, patches or static analysis reports, could be a sequence of method calls, a sequence of basic blocks or only one instruction. Note that not all target traces are actual traces, for instance a target trace containing dead code. Formally, we define the problem of directed fuzzing as:

Definition 6 (Directed fuzzing)**.** The goal of a *directed fuzzer* is, given a target t, to find a matching input s_0 for t.

Proposition 1. Let s be a reachable state. Let Σ be the callstack of s. Then, Σ is an instruction trace.

Proof. By construction. □

Proposition 2. Let s be a reachable state. Let ℓ be the current program location of s. Then, ℓ is an instruction execution trace.

Proof. By construction, and considering that program locations are isomorphic to sequences of length 1. □

3.1.2 Applications of Directed Fuzzing

Bug reproduction DGF is useful to reproduce disclosed bugs without the Proof-of-Concept (PoC). For example, due to concerns such as privacy, some applications (e.g., Microsoft's products) are not allowed to send the bug-triggering inputs. Thus, the developers can employ DGF to reproduce the crash based on the limited information provided, such as the method calls in the stack traces and some system configurations.

Patch testing A directed fuzzer can be used to test whether a patch is complete. Thus, directed fuzzing towards recent changes or patches has a higher chance of exposing newly-introduced bugs or incomplete bug fixes.

Static analysis report verification Static analysis can be leveraged to limit the search space in the testing and enhance directedness. In this setting, DGF can generate test inputs that show the vulnerability if it actually exists.

Information flow detection To detect data leakage vulnerabilities, a directed fuzzer can be used to generate executions that exercise sensitive sources containing private information and sensitive sinks where data becomes visible to the outside world.

Knowledge involvement It is possible to leverage the knowledge from developers or other techniques to provide more information to. For example, developers can help to identify the critical modules or potentially buggy functions based on the previous experience to drive fuzzing toward vulnerable parts.

3.1.3 Differences between Directed and Coverage-based Fuzzing

Target selection For DGF, a set of target locations must be identified manually or automatically in advance to guide the fuzzing process. Therefore, the target selection has a high impact on the performance of DGF. For example, selecting critical sites, such as `malloc()` and `free()`, as targets is more likely to allow DGF to detect heap-based memory corruption bugs.

Seed selection Since CGF aims to maximize the code coverage, CGF only retains inputs covering new paths and prioritizes an input simply based on its execution trace (e.g., quicker executions, larger traces, etc.). In contrast, DGF aims to reach specific predefined targets, it therefore prioritizes seeds that are "closer" to the targets using distance-based seed metric.

Exploration-exploitation For DGF, the whole fuzzing process is divided into two phases: the exploration phase and the exploitation phase. In the exploration phase, like existing coverage-guided fuzzers, DGF aims to explore as many paths as possible. Then, in the exploitation phase, DGF gives more chances of mutation to "closer" seeds that are more likely to generate inputs to reach the target. The intuition is that we should gradually assign more "energy" to a seed that is "closer" to the targets than to a seed that is "further away".

Triage In some settings such as bug reproduction, we need to verify whether a directed fuzzer triggers the expected bug with the expected stack traces in the triage step. Differently, for CGF, all unique crashing inputs are interesting.

3.2 Overview

3.2.1 Workflow

Figure 3.1 depicts the workflow of DGF. Overall directed fuzzers are built upon three main steps: (1) *instrumentation* (distance pre-computation), (2) *fuzzing* (including seed selection, power schedule and seed mutation) and (3) *triage*.

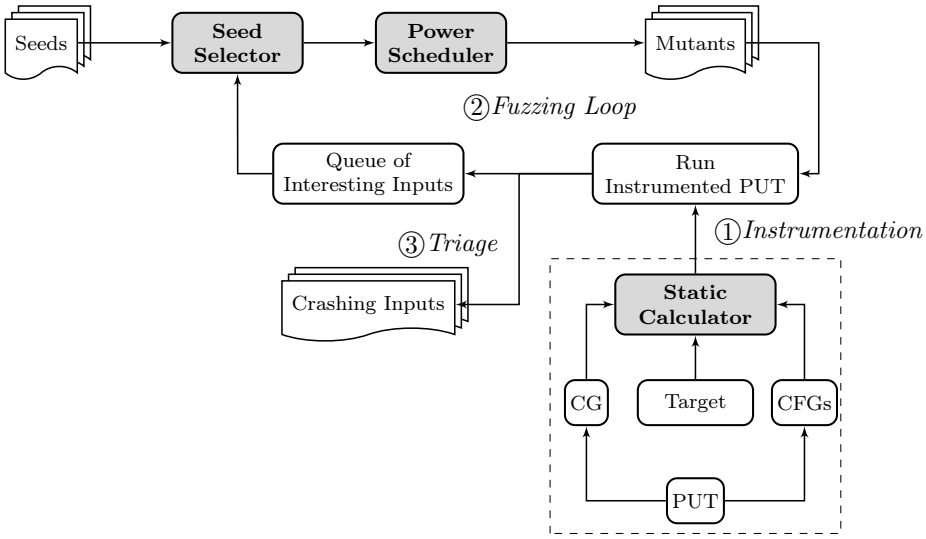


Figure 3.1: Workflow of DGF (different components compared to CGF are in gray).

3.2.2 Core Algorithm

The standard core algorithm of DGF is presented in Algorithm 1 (the different parts compared to CGF are in gray). Given a program P, a set of initial seeds S_0 and a target T, the algorithm outputs a set of bug-triggering inputs S'. The fuzzing queue S is initialized with the initial seeds in S_0 (line 1).

1. DGF first performs a static analysis (e.g., *target distance computation* for each basic block) and insert the instrumentation for dynamic coverage or distance information (line 2);

2. The fuzzer then repeatedly mutates inputs s chosen from the fuzzing queue S (line 4) until a timeout is reached. An input is selected either if it is *favored* (i.e., believed to be interesting) or with a small probability α (line 5). Subsequently, DGF assigns the *energy* (a.k.a, the number M of mutants to be created) to the selected seed s (line 6). Then, the fuzzer generates M new inputs by randomly applying some predefined mutation operators on seed s (line 8) and monitors their executions (line 9). If the generated mutant s' crashes the program, it is added to the set S' of crashing inputs (line 11). Also, newly generated mutants are added to the fuzzing queue[1] (line 13);

3. Finally, DGF returns S' as the set of bug-triggering inputs (triage does nothing in standard DGF) (line 14).

[1]This is a general view. In practice, seeds regarded as very uninteresting are already discarded at this point.

Algorithm 1: Directed Greybox Fuzzing

Input : Program P; Initial seeds S_0; Target locations T
Output: Bug-triggering seeds S'

```
1  S' := ∅; S := S_0;                                   ▷ S: the fuzzing queue
2  P' ← PREPROCESS(P, T) ;                              ▷ phase 1: Instrumentation
3  while timeout not exceeded do                        ▷ phase 2: Fuzzing
4      for s ∈ S do
5          if IS_FAVORED(s) or rand() ≤ α then
               ▷ seed selection, α: small probability
6              M := ASSIGN_ENERGY(s) ;                  ▷ power schedule
7              for i ∈ 1 ... M do
8                  s' := mutate_input(s);               ▷ seed mutation
9                  res := run(P', s', T);
10                 if is_crash(res) then
11                     S' := S' ∪ {s'};                 ▷ crashing inputs
12                 else
13                     S := S ∪ {s'};
14 S' = triage(S, S');                                  ▷ phase 3: Triage
15 return S';
```

3.3 Input Metrics

AFLGo [BPNR17] was the first to propose a CGF-based distance to evaluate the proximity between a seed execution and multiple targets, together with a simulated annealing-based power schedule. Hawkeye [CXL+18] keeps the CGF-based view but improves its accuracy[2], brings a seed selection heuristic partly based on target coverage (seen as a set of locations) and proposes adaptive mutations. In the following we describe in detail how existing directed fuzzers compute the most important score which is the seed distance.

3.3.1 Distance metric

Function level distance We define $d_f(n, T_f)$ as follow:

$$d_f(n, T_f) = \begin{cases} \textit{undefined} & \text{if } R(n, T_f) = \emptyset \\ \left[\sum_{t_f \in R(n,T_f)} d_f(n, t_f)^{-1}\right]^{-1} & \text{otherwise} \end{cases} \tag{3.1}$$

where $d_f(n, t_f)$ is the Dijkstra shortest distance between two functions n and t_f.

[2]Possibly at the price of both higher pre-computation costs due to more precise static analysis and runtime overhead due to complex seed metrics.

(a) Arithmetic Mean

(b) Harmonic Mean

Figure 3.2: Difference between node distance defined in terms of arithmetic mean versus harmonic mean. Node distance is shown in the white circles. The targets are marked in gray [BPNR17].

Basic block level distance We define $d_b(m, T_b)$ as follows:

$$d_b(m, T_b) = \begin{cases} 0 & \text{if } m \in T_b \\ c \cdot \min\limits_{n \in N(m)} (d_f(n, T_f)) & \text{if } m \in T \\ \left[\sum\limits_{t \in T} (d_b(m, t) + d_b(t, T_b))^{-1} \right]^{-1} & \text{otherwise} \end{cases} \tag{3.2}$$

where $d_b(m, t)$ is the Dijkstra shortest distance between two basic blocks m and t; $d_f(n, T_f)$ is the function level distance between function n and T_f in the call graph; $N(m)$ is the set of functions called by basic block m such that $\forall n \in N(m).R(n, T_f) \neq \emptyset$ where $R(n, T_f)$ is the set of all target functions that are reachable from n in Call Graph (CG); T is the set of basic blocks in CGF such that $\forall m \in T.N(m) \neq \emptyset$; c is a constant approximating the length of a trace between two functions. The harmonic mean allows to better measure the distance between two nodes to multiple targets, as illustrated in Figure 3.2.

Seed distance Let $\xi(s)$ be the execution trace of a seed s containing the exercised basic blocks. The distance $d_s(s, T_b)$ of a seed s to T_b as

$$d_s(s, T_b) = \frac{\sum_{m \in \xi(s)} d_b(m, T_b)}{|\xi(s)|} \tag{3.3}$$

Call graph edge weight While AFLGo uses the original CG whose edge weight w_{AFLGo} is always 1, Hawkeye proposes the Augmented Adjacent-Function Distance (AAFD) by augmenting the edge weight $w_{Hawkeye}$ based on the immediate call relation between the caller and the callee. For example, if f_b appears in both if and else branches in f_a as shown in Listing 3.1, $d_f(f_a, f_b)$ should be smaller than $d_f(f_a, f_c)$ if there is only one call of f_c in

f_a (same for Listing 3.2).

```
void fa (int i) {
  if (i > 0) {
    fb(i);
  } else {
    fb(i * 2);
    fc();
  }
}
```

Listing 3.1: Call pattern 1.

```
void fa (int i) {
  if(i > 0) {
    fb(i);
    fb(i * 2);
  } else {
    fc();
  }
}
```

Listing 3.2: Call pattern 2.

3.3.2 Covered function similarity metric

Furthermore, HAWKEYE also proposes the covered function similarity metric which measures the similarity between the seed execution trace and the target execution trace on the function level. The intuition is that seeds covering more functions in the expected bug trace will have more chances to be mutated to reach the targets.

Finally, HAWKEYE employs a fairness of traces based power schedule, that is calculated based on those seed metrics, to balance the effect of shorter traces and the longer traces that could reach the targets.

3.4 Differences between Source- and Binary-based Directed Fuzzing

Target locations We can extract target locations from the bug reports produced by the profiling tools, such as AddressSanitizer (ASan) or VALGRIND at source and binary level respectively. Different levels of analysis have different formats of target locations. Concretely, while the target location is represented in the format "`source_file:line_of_code`" at source level, binary-based DGF takes targets in the format "`function:address_of_block`". In both cases, different formats have the similar goal of representing a function call appeared in the stack traces of the bug report. Furthermore, it is clear that manually providing target locations at source level (e.g., only by reading the source code without running any tools like a disassembler) is a bit easier for users than at binary level.

Preprocessing It is worth noting that all proposed seed metrics, including the most important distance-based one, have some computations on CG and CGFs of the tested program. Different levels of analysis have different methods of pre-computing the static information. While existing source-based DGF currently relies on Low-Level Virtual Machine (LLVM)'s analysis tools to generate graphs, binary-based directed fuzzers first employ a binary disassembler, for example IDA PRO, to obtain those important graphs of the tested binary in the preprocessing phase.

Triaging Different levels of analysis rely on different profiling tools to triage inputs produced by the fuzzer in the final step. In practice, we often instrument the Program Under Test (PUT) with ASan such that source-based fuzzers can detect memory corruption bugs when they are triggered instead of silently corrupting some memory region. In case where the source code of PUT is not available, we employ VALGRIND, which is the state-of-the-art binary-only memory checker in the triaging phase. To the best of our knowledge, there is no practical approach to fuzz binaries with VALGRIND `Memcheck` due to its heavyweight instrumentation. Therefore, VALGRIND is more suitable for triaging inputs.

3.5 Limitations & Future Directions

Limitations DGF's limitations inherit from CGF as both techniques share the same workflow and key fuzzing components. Apart from those similar problems, such as complex input structures or hard-to-detect bugs, as discussed in §2.2.3, DGF has its own limitations that come from its existing solutions and implementation. We will discuss in brief DGF's significant limitations, some promising directions and revisit the thesis's main goals to close gaps in the state-of-the-art DGF.

1. **High instrumentation overhead** The distance computation is the first step in DGF's workflow. However, apart from some implementation issues [afl20f] of AFLGO, this process takes too long in some cases (e.g., several hours for Binutils) to calculate the distances and instrument them before use. Also, we may need to re-compute distances and instrument them again each time the targets are changed.

2. **Incomplete graphs** As distance metric plays an important role in examining the affinity between current input and the targets, the accuracy of Control Flow Graph (CFG) and especially CG majorly affect the calculation of the trace distance and also the whole fuzzing process. However, these graphs extracted at both source- or binary-level are incomplete due to indirect calls or indirect jumps. Dynamically updating the graphs at runtime may boost the fuzzing performance.

3. **Binary-level support** Existing directed fuzzers are mostly source-based approaches. One of the very few binary-based directed fuzzers is 1DVUL [PLL+19] that discovers 1-day vulnerabilities via binary patches by leveraging a hybrid approach of distance-based directed fuzzing and dominator-based directed symbolic execution. Developing fuzzers that are able to handle binary code in different security applications becomes increasingly necessary.

4. **Human-in-the-loop** Overall it is not easy for testers or developers to intervene with the directed fuzzing process. First, for the source-based directed fuzzers, the target (lines of code, e.g., "`main.c:10`") could be manually provided by users in some cases. However, for binary-based directed fuzzers, it is more tedious and challenging as the target is now a set of virtual addresses. Second, like CGF, users still have difficulties

in controlling the directed fuzzing at runtime, for example select targets back and forth without stopping the fuzzing process since existing directed fuzzers may need to repeat the instrumentation process when modifying the targets.

3.6 Conclusion

In this chapter, we present a survey of DGF. By introducing the core techniques of recent DGF in many aspects, we hope to provide a background for readers to follow the later technical chapters.

Revisit our goal Having described in brief the existing automated vulnerability detection methods, their classification based on search space exploration techniques, practical dynamic approaches like (directed) greybox fuzzing and their problems, we finally turn our focus towards designing a solution to address some of the aforementioned limitations.

Our goal is to *develop effective directed fuzzing techniques to detect complex typestate vulnerabilities, such as Use-After-Free (UAF), at binary level in diverse security applications with a low overhead.*

Chapter 4

Binary-level Directed Fuzzing for Use-Afer-Free Vulnerabilities

Contents

4.1 Introduction **34**
4.2 Motivation **35**
4.3 The UAFuzz Approach **37**
4.3.1 Bug Trace Flattening 38
4.3.2 Seed Selection based on Target Similarity 40
4.3.3 UAF-based Distance 42
4.3.4 Power Schedule 44
4.3.5 Postprocess and Bug Triage 46
4.4 Experimental Evaluation **47**
4.4.1 Research Questions 47
4.4.2 Evaluation Setup 47
4.4.3 UAF Bug-reproducing Ability (RQ1) 49
4.4.4 UAF Overhead (RQ2) 53
4.4.5 UAF Triage (RQ3) 56
4.4.6 Individual Contribution (RQ4) 57
4.4.7 Patch Testing & Zero-days 58
4.4.8 Threats to Validity 61
4.5 Related Work **62**
4.5.1 Directed Greybox Fuzzing 62
4.5.2 Coverage-based Greybox Fuzzing 62
4.5.3 UAF Detection 62
4.5.4 UAF Fuzzing Benchmark 63
4.6 Conclusion **64**

In this chapter, we introduce UAFuzz, a binary-level directed fuzzer specializing to detect CWE-415 Double-Free (DF) and CWE-416 Use-After-Free (UAF). We evaluate the effectiveness and efficiency of our proposed techniques on real-world programs against the state-of-the-art (directed) greybox fuzzers via several research questions. UAFuzz is also

applicable in both applications: bug reproduction and patch testing.

4.1 Introduction

Proposal We propose UAFuzz, *the first (binary-level) directed greybox fuzzer tailored to UAF bugs.* A quick comparison of UAFuzz with existing greybox fuzzers in terms of UAF is presented in Table 4.1. While we follow mostly the generic scheme of directed fuzzing, we carefully tune several of its key components to the specifics of UAF:

- the *distance metric* favors shorter call chains leading to the target functions that are more likely to include both allocation and free functions – where state-of-the-art directed fuzzers rely on a generic Control Flow Graph (CFG)-based distance;
- *seed selection* is now based on a *sequenceness-aware target similarity metric* – where state-of-the-art directed fuzzers rely at best on target coverage;
- our *power schedule* benefits from these new metrics, plus another one called *cut-edges* favoring prefix paths more likely to reach the whole target.

Table 4.1: Summary of existing greybox fuzzing techniques.

	AFL	AFLGo	Hawkeye	UAFuzz
Directed fuzzing approach	✗	✓	✓	✓
Support binary	✓	✗	✗	✓
UAF bugs oriented	✗	✗	✗	✓
Fast instrumentation	✓	✗	✗	✓
UAF bugs triage	✗	✗	✗	✓

Finally, the bug triaging step piggy-backs on our previous metrics to pre-identifies seeds as likely-bugs or not, sparing a huge amount of queries to the profiling tool for confirmation (Valgrind [NS07] in our implementation).

Contributions Our contribution is the following:

- We design the first directed greybox fuzzing technique *tailored to* UAF bugs directed fuzzing (selection heuristic, power schedule, input metrics) and specialize them to UAF. These improvements are proven beneficial and complementary;
- We develop a toolchain on top of the state-of-the-art greybox fuzzer AFL [afl20a] and the binary analysis platform Binsec [bin20], named UAFuzz [uaf20b], implementing the above method for UAF directed fuzzing over binary codes and enjoying small overhead;

- We construct and openly release [uaf20a] the largest fuzzing benchmark dedicated to UAF, comprising **30** real bugs from **17** widely-used projects (including the few previous UAF bugs found by directed fuzzers), in the hope of facilitating future UAF fuzzing evaluation;
- We evaluate our technique and tool in a bug reproduction setting (Section 4.4), demonstrating that UAFuzz is highly effective and significantly outperforms state-of-the-art competitors: 2× faster in average to trigger bugs (up to 43×), +34% more successful runs in average (up to +300%) and 17× faster in triaging bugs (up to 130×, with 99% spare checks);
- Finally, UAFuzz is also proven effective in patch testing (§4.4.7), leading to the discovery of 32 unknown bugs (13 UAFs, 10 CVEs) in projects like GNU Binutils, GPAC, MuPDF and GNU Patch (including 4 buggy patches). So far, 17 have been fixed.

UAFuzz is the *first* directed greybox fuzzing approach tailored to detecting UAF vulnerabilities (in binary) given only bug stack traces. UAFuzz outperforms existing directed fuzzers on this class of vulnerabilities for bug reproduction and encouraging results have been obtained as well on patch testing. We believe that our approach may also be useful in slightly related contexts, for example partial bug reports from static analysis or other classes of vulnerabilities.

4.2 Motivation

The toy example in Listing 4.1 contains a UAF bug due to a missing `exit()` call, a common root cause in such bugs (e.g., CVE-2014-9296, CVE-2015-7199). The program reads a file and copies its contents into a buffer `buf`. Specifically, a memory chunk pointed at by `p` is allocated (line 12), then `p_alias` and `p` become aliased (line 15). The memory pointed by both pointers is freed in function `bad_func` (line 11). The UAF bug occurs when the freed memory is dereferenced again via `p` (line 19).

Bug-triggering conditions The UAF bug is triggered iff the first three bytes of the input are '`AFU`'. To quickly detect this bug, fuzzers need to explore the right path through the `if` part of conditional statements in lines 13, 5 and 18 in order to cover in sequence the three UAF events *alloc*, *free* and *use* respectively. It is worth noting that this UAF bug does not make the program crash, hence existing greybox fuzzers without sanitization will not detect this memory error.

Coverage-based greybox fuzzing Starting with an empty seed, AFL quickly generates 3 new inputs (e.g., '`AAAA`', '`FFFF`' and '`UUUU`') triggering individually the 3 UAF events. None of these seeds triggers the bug. As the probability of generating an input starting with '`AFU`' from an empty seed is extremely small, the coverage-guided mechanism is not

```
1  int *p, *p_alias;
2  char buf[10];
3  void bad_func(int *p) {free(p);} /* exit() is missing */
4  void func() {
5    if (buf[1] == 'F')
6      bad_func(p);
7    else /* lots more code ... */
8  }
9  int main (int argc, char *argv[]) {
10   int f = open(argv[1], O_RDONLY);
11   read(f, buf, 10);
12   p = malloc(sizeof(int));
13   if (buf[0] == 'A'){
14     p_alias = malloc(sizeof(int));
15     p = p_alias;
16   }
17   func();
18   if (buf[2] == 'U')
19     *p = 1;
20   return 0;
21 }
```

Listing 4.1: Motivating example.

effective here in tracking a sequence of UAF events even though each individual event is easily triggered.

Directed greybox fuzzing Given a bug trace (14 – *alloc*, 17, 6, 3 – *free*, 19 – *use*) generated for example by ASan, Directed Greybox Fuzzing (DGF) prevents the fuzzer from exploring undesirable paths, e.g., the `else` part at line 7 in function `func`, in case the condition at line 5 is more complex. Still, directed fuzzers have their own blind spots. For example, standard DGF seed selection mechanisms favor a seed whose execution trace covers many locations in targets, instead of trying to reach these locations in a given order. For example, regarding a target (A, F, U), standard DGF distances [BPNR17, CXL+18] do not discriminate between an input s_1 covering a path $A \to F \to U$ and another input s_2 covering $U \to A \to F$. The lack of ordering in exploring target locations makes UAF bug detection very challenging for existing directed fuzzers. Another example: the power function proposed by HAWKEYE [CXL+18] may assign much energy to a seed whose trace does not reach the target function, implying that it could get lost on the toy example in the `else` part at line 7 in function `func`.

A glimpse at UAFUZZ We rely in particular on modifying the seed selection heuristic w.r.t. the number of targets covered by an execution trace (§4.3.2) and bringing target ordering-aware seed metrics to DGF (§4.3.3).

On the toy example, UAFUZZ generates inputs to progress towards the expected target sequences. For example, in the same fuzzing queue containing 4 inputs, the mutant '`AFAA`', generated by mutating the seed '`AAAA`', is discarded by AFL as it does not increase code

coverage. However, since it has maximum value of target similarity metric score (i.e., 4 targets including lines 14, 17, 6, 3) compared to all 4 previous inputs in the queue (their scores are 0 or 2), this mutant is selected by UAFuzz for subsequent fuzzing campaigns. By continuing to fuzz '`AFAA`', UAFuzz eventually produces a bug-triggering input, e.g., '`AFUA`'.

Evaluation AFLGo *(source-level) cannot detect the UAF bug within 2 hours*[1][2]*, while* UAFuzz *(binary-level) is able to trigger it within 20 minutes. Also,* UAFuzz *sends a single input to* Valgrind *for confirmation (the right Proof-of-Concept (PoC) input), while* AFLGo *sends 120 inputs.*

4.3 The UAFuzz Approach

UAFuzz is made out of several components encompassing seed selection (§4.3.2), input metrics (§4.3.3), power schedule (§4.3.4), and seed triage (§4.3.5). Before detailing these aspects, let us start with an overview of the approach.

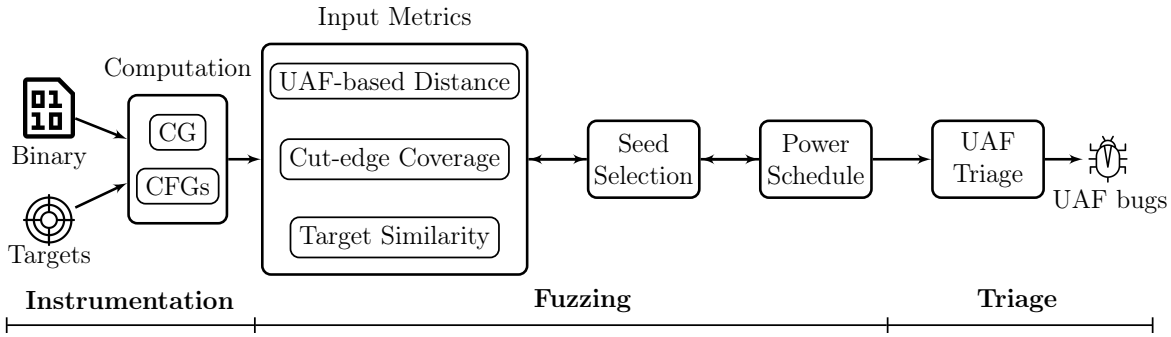


Figure 4.1: Overview of UAFuzz.

We aim to find an input fulfilling both control-flow (temporal) and runtime (spatial) conditions to trigger the UAF bug. We solve this problem by bringing UAF characteristics into DGF in order to generate more potential inputs reaching targets in sequence w.r.t. the UAF expected bug trace. Figure 4.1 depicts the general picture. Especially:

- We propose three dynamic seed metrics specialized for UAF vulnerabilities detection. The distance metric approximates how close a seed is to all target locations (§4.3.3), and takes into account the need for the seed execution trace to cover the three UAF events in order. The cut-edge coverage metric (§4.3.4.1) measures the ability of a seed to take the correct decision at important decision nodes. Finally, the target similarity metric concretely assesses how many targets a seed execution trace covers at runtime (§4.3.2.2);

[1] AFL-QEMU did not succeed either.
[2] Hawkeye is not available and thus could not be tested.

- Our seed selection strategy (§4.3.2) favors seeds covering more targets at runtime. The power scheduler determining the energy for each selected seed based on its metric scores during the fuzzing process is detailed in §4.3.4;
- Finally, we take advantage of our previous metrics to pre-identify likely-PoC inputs that are sent to a profiling tool (here VALGRIND) for bug confirmation, avoiding many useless checks (§4.3.5).

4.3.1 Bug Trace Flattening

Bug trace As stack traces provide (partial) information about the sequence of program locations leading to a crash, they are extremely valuable for bug reproduction [JO12, BPNR17, CXL+18, LZY+19]. Yet, as crashes caused by UAF bugs may happen long after the UAF happened, standard stack traces usually do not help in reproducing UAF bugs. Hopefully, profiling tools for dynamically detecting memory corruptions, such as ASan [SBPV12] or VALGRIND [NS07], record the stack traces of *all memory-related events*: when they detect that an object is used after being freed, they actually report *three stack traces* (when the object is allocated, when it is freed and when it is used after being freed). *We call such a sequence of three stack traces a* ***UAF bug trace****.* When we use a bug trace as an input to try to reproduce the bug, we call such a bug trace a *target*.

```
    // Stack trace for the bad Use
==4440== Invalid read of size 1
==4440== at 0x40A8383: vfprintf (vfprintf.c:1632)
==4440== by 0x40A8670: buffered_vfprintf (vfprintf.c:2320)
==4440== by 0x40A62D0: vfprintf (vfprintf.c:1293)
==4440== by 0x80AA58A: error (elfcomm.c:43)
==4440== by 0x8085384: process_archive (readelf.c:19063)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==4440== by 0x8085C6E: main (readelf.c:19318)

    // Stack trace for the Free
==4440== Address 0x421fdc8 is 0 bytes inside a block of size 86 free'd
==4440== at 0x402D358: free (in vgpreload_memcheck-x86-linux.so)
==4440== by 0x80857B4: process_archive (readelf.c:19178)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==4440== by 0x8085C6E: main (readelf.c:19318)

    // Stack trace for the Alloc
==4440== Block was alloc'd at
==4440== at 0x402C17C: malloc (in vgpreload_memcheck-x86-linux.so)
==4440== by 0x80AC687: make_qualified_name (elfcomm.c:906)
==4440== by 0x80854BD: process_archive (readelf.c:19089)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==4440== by 0x8085C6E: main (readelf.c:19318)
```

Figure 4.2: Bug trace of CVE-2018-20623 (UAF) produced by VALGRIND.

A bug trace is a sequence of stack traces, i.e. it is a large object not fit for the

Algorithm 2: Bug Trace Flattening

Let $\langle \Sigma_1, \ldots, \Sigma_n \rangle$ be a callstack trace. The algorithm has two steps:

- Consider all callstacks as paths in a tree, and reconstitute the tree corresponding to these paths (the dynamic call tree)
- Traverse the tree in preorder.

For instance, the flattening of $\langle \langle \ell_1, \ell_2, \ell_3 \rangle, \langle \ell_1, \ell_2, \ell_4 \rangle \rangle$ is $\langle \ell_1, \ell_2, \ell_3, \ell_4 \rangle$.

lightweight instrumentation required by greybox fuzzing. The most valuable information that we need to extract from a bug trace is the sequence of basic blocks (and functions) that were traversed, which is an easier object to work with. We call this extraction *bug trace flattening*. The operation works as follows. First, each of the three stack-traces is seen as a path in a call tree; we thus merge all the stack traces to re-create that tree. Some of the nodes in the tree have several children; we make sure that the children are ordered according to the ordering of the UAF events (i.e. the child coming from the *alloc* event comes before the child coming from the *free* event). Figure 4.3 shows an example of a tree for the bug trace given in Figure 4.2.

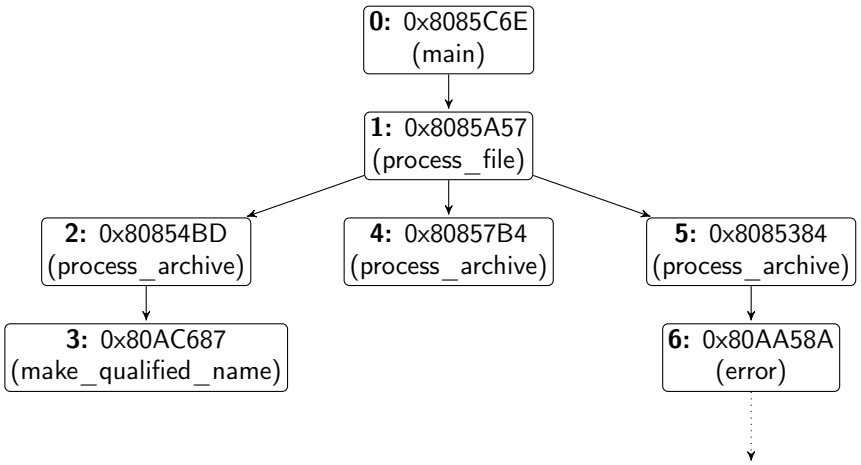


Figure 4.3: Reconstructed Dynamic Calling Tree (DCT) from CVE-2018-20623 (bug trace from Figure 4.2). The preorder traversal of this tree is simply $0 \to 1 \to 2 \to 3(n_{alloc}) \to 4(n_{free}) \to 5 \to 6(n_{use})$.

Proposition 3. The result of flattening a callstrack trace is an instruction trace.

Proof. It suffices to show that the real execution going through the callstack trace also goes through the instruction trace. This is not that trivial: for instance, a naive flattening that would flatten $\langle \langle \ell_1, \ell_2, \ell_3 \rangle, \langle \ell_1, \ell_2, \ell_4 \rangle \rangle$ into $\langle \ell_1, \ell_2, \ell_3, \ell_1, \ell_2, \ell_4 \rangle$ could not result in an instruction trace.

□

Proposition 4. Let the initial state s_0 and the target callstack trace T such that s_0 matches T (or $s_0 \vDash T$). Let the instruction trace t be the flattening of T. Then, $s_0 \vDash t$.

Proposition 5. There exists s_0, t, T such that t is the flattening of T, $s_0 \vDash t$ and $s_0 \nvDash T$.

Proof. This can happen even if we flatten a single callstack. Consider the following program:

```
void f1(int a){ l1: f2(a); if(a == 1) { l2: f3(); }}
void f2(int a){ if(a == 0){ l3:f3();} }
void f3() { l4:assert(false); }
```

If the target callstack trace is $T = \langle\langle \ell_1, \ell_3, \ell_4 \rangle\rangle$, then only the input `a==0` will match this target. But if we flatten it to an instruction trace $t = \langle \ell_1, \ell_3, \ell_4 \rangle$, then both the inputs `a==0` and `a==1` will match the target. □

The propositions above mean that callstack traces are more precise targets than the corresponding intruction traces. Finally, we perform a preorder traversal of this tree to get a sequence of target locations (and their associated functions), which will used to guide the fuzzer at runtime in the following algorithms.

4.3.2 Seed Selection based on Target Similarity

Fuzzers generate a large number of inputs so that smartly selecting the seed from the fuzzing queue to be mutated in the next fuzzing campaign is crucial for effectiveness. Our seed selection algorithm is based on two insights. First, we *should prioritize seeds that are most similar to the target bug trace*, as the goal of a directed fuzzer is to find bugs covering the target bug trace. Second, *target similarity should take ordering (a.k.a. sequenceness) into account*, as traces covering sequentially a number of locations in the target bug trace are closer to the target than traces covering the same locations in an arbitrary order.

4.3.2.1 Seed Selection

Definition 7 (Max-reaching input)**.** A *max-reaching input* is an input s whose execution trace is the most similar to the target bug trace T so far, where most similar means "having the highest value as compared by a target similarity metric $t(s, T)$".

Algorithm 3: IS_FAVORED

Input : A seed s
Output: *true* if s is favored, otherwise *false*

```
1 global t_max = 0;                          ▷ maximum target similar metric score
2 if t(s) ≥ t_max then t_max = t(s); return true;   ▷ update t_max
3 else if new_cov(s) then return true;       ▷ increase coverage
4 else return false;
```

We mostly select and mutate max-reaching inputs during the fuzzing process. Nevertheless, we also need to improve code coverage, thus UAFuzz also selects inputs that cover new paths, with a small probability α (Algorithm 1). In our experiments, the probability of selecting the remaining inputs in the fuzzing queue that are less favored is 1% like AFL [afl20a].

4.3.2.2 Target Similarity Metrics

A *target similarity metric* $t(s, T)$ measures the similarity between the execution of a seed s and the target UAF bug trace T. We define 4 such metrics, based on whether we consider ordering of the covered targets in the bug trace (P), or not (B) – P stands for Prefix, B for Bag; and whether we consider the full trace, or only the three UAF events ($3T$):

- Target prefix $t_P(s, T)$: locations in T covered in sequence by executing s until first divergence;
- UAF prefix $t_{3TP}(s, T)$: UAF events of T covered in sequence by executing s until first divergence;
- Target bag $t_B(s, T)$: locations in T covered by executing s;
- UAF bag $t_{3TB}(s, T)$: UAF events of T covered by s.

For example, using Listing 4.1, the 4 metric values of a seed s '`ABUA`' w.r.t. the UAF bug trace T are: $t_P(s, T) = 2$, $t_{3PT}(s, T) = 1$, $t_B(s, T) = 3$ and $t_{3TB}(s, T) = 2$.

These 4 metrics have different degrees of *precision*. A metric t is said *more precise than* a metric t' if, for any two seeds s_1 and s_2: $t(s_1, T) \geq t(s_2, T) \Rightarrow t'(s_1, T) \geq t'(s_2, T)$. Figure 4.4 compares our 4 metrics w.r.t their relative precision.

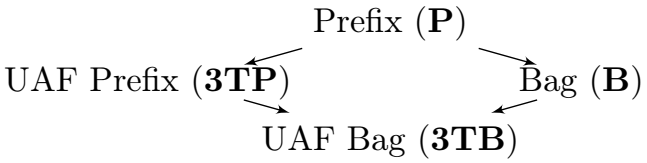


Figure 4.4: Precision lattice for Target Similarity Metrics.

4.3.2.3 Combining Target Similarity Metrics

Using a precise metric such as P allows to better assess progression towards the goal. In particular, P can distinguish seeds that match the target bug trace from those that do not, while other metrics cannot. On the other hand, a less precise metric provides information that precise metrics do not have. For instance, P does not measure any difference between traces whose suffix would match the target bug trace, but who would diverge from the target trace on the first locations (like '`UUU`' and '`UFU`' on Listing 4.1), while B can.

To take benefit from both precise and imprecise metrics, we combine them using a lexicographical order. Hence, the *P-3TP-B* metric is defined as:

$$t_{P-3TP-B}(s,T) \triangleq \langle t_P(s,T), t_{3TP}(s,T), t_B(s,T) \rangle \tag{4.1}$$

This combination favors first seeds that cover the most locations in the prefix, then (in case of tie) those reaching the most number of UAF events in sequence, and finally (in case of tie) those that reach the most locations in the target. Based on preliminary investigation, we default to *P-3TP-B* for seed selection in UAFuzz.

4.3.3 UAF-based Distance

One of the main component of directed greybox fuzzers is the computation of a *seed distance*, which is an evaluation of a distance from the execution trace of a seed s to the target. The main heuristic here is that if the execution trace of s is close to the target, then s is close to an input that would cover the target, which makes s an interesting seed. In existing directed greybox fuzzers [afl20e, CXL+18], the seed distance is computed to a target which is a single location or a set of locations. This is not appropriate for the reproduction of UAF bugs, that must go through 3 different locations in sequence. Thus, we propose to modify the seed distance computation to take into account the need to reach the locations in order.

4.3.3.1 Zoom: Background on Seed Distances

Existing directed greybox fuzzers [afl20e, CXL+18] compute the distance $d(s,T)$ from a seed s to a target T as follows.

AFLGo's seed distance [afl20e] The *seed distance* $d(s,T)$ is defined as the (arithmetic) mean of the *basic-block distances* $d_b(m,T)$, for each basic block m in the execution trace of s. The *basic-block distance* $d_b(m,T)$ is defined using the length of the intra-procedural shortest path from m to the basic block of a "call" instruction, using the CFG of the function containing m; and the length of the inter-procedural shortest path from the function containing m to the target functions T_f (in our case, T_f is the function where the *use* event happens), using the call graph.

Hawkeye's enhancement [CXL+18] The main factor in this seed distance computation is computing distance between functions in the call graph. To compute this, AFLGo uses the original call graph with every edge having weight 1. Hawkeye improves this computation by proposing the augmented adjacent-function distance (AAFD), which changes the edge weight from a caller function f_a and a callee f_b to $w_{Hawkeye}(f_a, f_b)$. The idea is to favor edges in the call graph where the callee can be called in a variety of situations, i.e. appear several times at different locations.

4.3.3.2 Our UAF-based Seed Distance

Previous seed distances [afl20e, CXL+18] do not account for any order among the target locations, while it is essential for UAF. We address this issue by modifying the distance between functions in the call graph to favor paths that *sequentially* go through the three UAF events *alloc*, *free* and *use* of the bug trace. This is done by decreasing the weight of the edges in the call graph that are likely to be between these events, using lightweight static analysis.

This analysis first computes R_{alloc}, R_{free}, and R_{use}, i.e., the sets of functions that can call respectively the *alloc*, *free*, or *use* function in the bug trace – the *use* function is the one where the *use* event happens. Then, we consider each call edge between f_a and f_b as indicating a direction: either downward (f_a executes, then calls f_b), or upward (f_b is called, then f_a is executed). Using this we compute, for each direction, how many events in sequence can be covered by going through the edge in that direction. For instance, if $f_a \in R_{alloc}$ and $f_b \in R_{free} \cap R_{use}$, then taking the $f_a \to f_b$ call edge possibly allows to cover the three UAF events in sequence. To find double free, we also include, in this computation, call edges that allow to reach two *free* events in sequence.

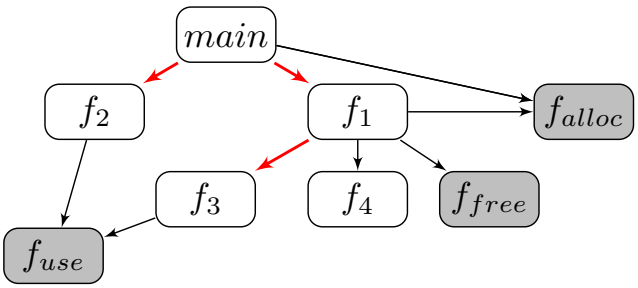


Figure 4.5: Example of a call graph. Favored edges are in red.

Then, we favor a call edge from f_a to f_b by decreasing its weight, based on how many events in sequence the edge allows to cover. Figure 4.5 presents an example of call graph with edges favored using the above Θ_{UAF} function. In our experiments, we use the following $\Theta_{UAF}(f_a, f_b)$ function, with a value of $\beta = 0.25$:

$$\Theta_{UAF}(f_a, f_b) \triangleq \begin{cases} \beta & \text{if } f_a \to f_b \text{ covers more than 2 UAF events in sequence} \\ 1 & \text{otherwise} \end{cases} \tag{4.2}$$

Finally, we combine our edge weight modification with that of HAWKEYE:

$$w_{UAFuzz}(f_a, f_b) \triangleq w_{Hawkeye}(f_a, f_b).\Theta_{UAF}(f_a, f_b) \tag{4.3}$$

Like AFLGO, we favor the shortest path leading to the targets, since it is more likely to involve only a small number of control flow constraints, making it easier to cover by fuzzing. Our distance-based technique therefore considers both calling relations in general, via $w_{Hawkeye}$, and calling relations covering UAF events in sequence, via Θ_{UAF}.

4.3.4 Power Schedule

Coverage-guided fuzzers employ a power schedule (or energy assignment) to determine the number of extra inputs to generate from a selected input, which is called the *energy* of the seed. It measures how long we should spend fuzzing a particular seed. While AFL [afl20a] mainly uses execution trace characteristics such as trace size, execution speed of the PUT and time added to the fuzzing queue for seed energy allocation, recent work [BPR16, RJK+17, LXC+19] including both directed and coverage-guided fuzzing propose different power schedules. AFLGo employs simulated annealing to assign more energy for seeds closer to target locations (using the seed distance), while Hawkeye accounts for both shorter and longer traces leading to the targets via a power schedule based on trace distance and similarity at function level.

We propose here a new power schedule using the intuitions that we should assign more energy to seeds in these cases:

- seeds that are closer (using the seed distance, §4.3.3.2);
- seeds that are more similar to the target (using the target similarity, §4.3.2.2);
- *seeds that make better decisions at critical code junctions* (we define hereafter a new metric to evaluate the latter case in §4.3.4.1).

4.3.4.1 Cut-edge Coverage Metric

To track progress of a seed during the fuzzing process, a fine-grained approach would consist in instrumenting the execution to compare the similarity of the execution trace of the current seed with the target bug trace, at the basic block level. But this method would slow down the fuzzing process due to high runtime overhead, especially for large programs. A more coarse-grained approach, on the other hand, is to measure the similarity at function level as proposed in Hawkeye [CXL+18]. However, a callee can occur multiple times from different locations of single caller. Also, reaching a target function does not mean reaching the target basic blocks in this function.

Thus, we propose the lightweight *cut-edge coverage metric*, hitting a middle ground between the two aforementioned approaches by measuring progress *at the edge level* but *on the critical decision nodes only*.

Definition 8 ((Non-) Cut edge)**.** A *cut edge* between two basic blocks *source* and *sink* is an outgoing edge of a decision node so that there exists a path starting from *source*, going through this edge and reaching *sink*. A *non-cut edge* is an edge which is not a cut-edge, i.e. for which there is no path from *source* to *sink* that go through this edge.

Algorithm 4 shows how cut/non-cut edges are identified in UAFuzz given a tested binary program and an expected UAF bug trace. The main idea is to identify and accumulate the cut edges between all consecutive nodes in the (flattened) bug trace. For instance in the bug trace of Figure 4.3, we would first compute the cut edges between 0

Algorithm 4: Accumulating cut edges

Input : Program P; dynamic calling tree T of a bug trace
Output: Set of cut edges E_{cut}

```
E_cut ← ∅;
nodes ← flatten(T);
for n ∈ nodes ∧ pn the node before n in T do
    if n.func == pn.func then
        ce ← calculate_cut_edges(n.func, pn.bb, n.bb);
    else if pn is a call to n.func then
        ce ← calculate_cut_edges(n.func, n.func.entry_bb, n.bb);
    E_cut ← E_cut ∪ ce;
return E_cut;
```

Algorithm 5: calculate_cut_edges inside a function

Input : A function f; Two basic blocks bb_{source} and bb_{sink} in f
Output: Set of cut edges ce

```
ce ← ∅;
cfg ← get_CFG(f);
decision_nodes ← {dn : ∃ a path bb_source →* dn →* bb_sink in cfg}
for dn ∈ decision_nodes do
    outgoing_edges ← get_outgoing_edges(cfg, dn);
    for edge ∈ outgoing_edges do
        if reachable(cfg, edge, bb_sink) then
            ce ← ce ∪ {edge};
return ce;
```

and 1, then those between 1 and 2, etc. As the bug trace is a sequence of stack traces, most of the locations in the trace are "call" events, and we compute the cut edge from the function entry point to the call event in that function. However, because of the flattening, sometimes we have to compute the cut edges between different points in the same function (e.g. if in the bug trace the same function is calling *alloc* and *free*, we would have to compute the edge from the call to *alloc* to the call to *free*).

Algorithm 5 describes how cut-edges are computed inside a single function. First we have to collect the decision nodes, i.e. conditional jumps between the source and sink basic blocks. This can be achieved using a simple data-flow analysis. For each outgoing edge of the decision node, we check whether they allow to reach the sink basic block; those that can are cut edges, and the others are non-cut edges. Note that this program analysis is intra-procedural, so that we do not need construct an inter-procedural CFG.

Our heuristic is that an input exercising more cut edges and fewer non-cut edges is more likely to cover more locations of the target. Let $E_{cut}(T)$ be the set of all cut edges of a program given the expected UAF bug trace T. We define the cut-edge score $e_s(s, T)$ of

seed s as

$$e_s(s,T) \triangleq \sum_{e \in E_{cut}(T)} \lfloor (\log_2 hit(e) + 1) \rfloor - \delta * \sum_{e \notin E_{cut}(T)} \lfloor (\log_2 hit(e) + 1) \rfloor \tag{4.4}$$

where $hit(e)$ denotes the number of times an edge e is exercised, and $\delta \in (0, 1)$ is the weight penalizing seeds covering non-cut edges. In our main experiments, we use $\delta = 0.5$ according to our preliminary experiments. To deal with the path explosion induced by loops, we use bucketing [afl20a]: the hit count is bucketized to small powers of two.

4.3.4.2 Energy Assignment

We propose a power schedule function that assigns energy to a seed using a combination of the three metrics that we have proposed: the prefix target similarity metric $t_P(s,T)$ (§4.3.2.2), the UAF-based seed distance $d(s,T)$ (§4.3.3.2), and the cut-edge coverage metric $e_s(s,T)$ (§4.3.4.1). The idea of our power schedule is to assign energy to a seed s proportionally to the number of targets covered in sequence $t_P(s,T)$, with a corrective factor based on seed distance d and cut-edge coverage e_s. Indeed, our power function (corresponding to ASSIGN_ENERGY in Algorithm 1) is defined as:

$$p(s,T) \quad \triangleq \quad (1 + t_P(s,T)) \ \times \ \tilde{e}_s(s,T) \ \times \ (1 - \tilde{d}_s(s,T)) \tag{4.5}$$

Because their actual value is not as meaningful as the length of the covered prefix, but they allow to rank the seeds, we apply a min-max normalization [afl20e] to get a *normalized seed distance* ($\tilde{d}_s(s,T)$) and *normalized cut-edge score* ($\tilde{e}_s(s,T)$). For example, $\tilde{d}_s(s,T) = \frac{d_s(s,T) - minD}{maxD - minD}$ where *minD, maxD* denote the minimum and maximum value of seed distance so far. Note that both metric scores are in (0, 1), i.e. can only reduce the assigned energy when their score is bad.

4.3.5 Postprocess and Bug Triage

Since UAF bugs are often silent, all seeds generated by a directed fuzzer must *a priori* be sent to a *bug triager* (typically, a profiling tool such as VALGRIND) in order to confirm whether they are bug triggering input or not. Yet, this can be extremely expensive as fuzzers generate a huge amount of seeds and bug triagers are expensive.

Fortunately, the target similarity metric allows UAFUZZ to compute the sequence of covered targets of each fuzzed input at runtime. This information is *available for free* for each seed once it has been created and executed. We capitalize on it in order to *pre-identify* likely-bug triggering seeds, i.e. seeds that indeed cover the three UAF events in sequence. Then, the bug triager is run only over these pre-identified seeds, the other ones being simply discarded – potentially saving a huge amount of time in bug triaging.

4.4 Experimental Evaluation

4.4.1 Research Questions

To evaluate the effectiveness and efficiency of our approach, we investigate four principal research questions:

RQ1. UAF Bug-reproducing Ability Can UAFuzz outperform other directed fuzzing techniques in terms of UAF bug reproduction in executables?

RQ2. UAF Overhead How does UAFuzz compare to other directed fuzzing approaches w.r.t. instrumentation time and runtime overheads?

RQ3. UAF Triage How much does UAFuzz reduce the number of inputs to be sent to the bug triage step?

RQ4. Individual Contribution How much does each UAFuzz component contribute to the overall results?

We will also evaluate UAFuzz in the context of ***patch testing***, another important application of directed fuzzing [BPNR17, CXL+18, PLL+19].

4.4.2 Evaluation Setup

Evaluation fuzzers We aim to compare UAFuzz with state-of-the-art directed fuzzers, namely AFLGo [afl20e] and Hawkeye [CXL+18], using AFL-QEMU as a baseline (binary-level coverage-based fuzzing). Unfortunately, both AFLGo and Hawkeye work on source code, and while AFLGo is open source, Hawkeye is not available. Hence, we *implemented binary-level versions* of AFLGo and Hawkeye, coined as AFLGoB and HawkeyeB. We closely follow the original papers, and, for AFLGo, use the source code as a reference. AFLGoB and HawkeyeB are implemented on top of AFL-QEMU, following the generic architecture of UAFuzz but with dedicated distance, seed selection and power schedule mechanisms. We discuss in details the implementation of UAFuzz in Chapter 5. Table 4.2 summarizes our different fuzzer implementations and a comparison with their original counterparts.

Table 4.2: Overview of main techniques of greybox fuzzers. Our own implementations are marked with ⋆.

Fuzzer	Directed	Binary?	Distance	Seed Selection	Power Schedule	Mutation
AFL-QEMU	✗	✓	–	AFL	AFL	AFL
AFLGo	✓	✗	CFG-based	∼ AFL	Annealing	∼ AFL
AFLGoB⋆	✓	✓	∼ AFLGo	∼ AFLGo	∼ AFLGo	∼ AFLGo
Hawkeye	✓	✗	AAFD	distance-based	Trace fairness	Adaptive
HawkeyeB⋆	✓	✓	∼ Hawkeye	∼ Hawkeye	≈ Hawkeye	∼ AFLGo
UAFuzz⋆	✓	✓	UAF-based	Targets-based	UAF-based	∼ AFLGo

Table 4.3: Overview of our evaluation benchmark.

Bug ID	Program		Bug		#Targets in trace
	Project	Size	Type	Crash	
giflib-bug-74	GIFLIB	59 Kb	DF	✗	7
CVE-2018-11496	lrzip	581 Kb	UAF	✗	12
yasm-issue-91	yasm	1.4 Mb	UAF	✗	19
CVE-2016-4487	Binutils	3.8 Mb	UAF	✓	7
CVE-2018-11416	jpegoptim	62 Kb	DF	✗	5
mjs-issue-78	mjs	255 Kb	UAF	✗	19
mjs-issue-73	mjs	254 Kb	UAF	✗	28
CVE-2018-10685	lrzip	576 Kb	UAF	✗	7
CVE-2019-6455	Recutils	604 Kb	DF	✗	15
CVE-2017-10686	NASM	1.8 Mb	UAF	✓	10
gifsicle-issue-122	Gifsicle	374 Kb	DF	✗	11
CVE-2016-3189	bzip2	26 Kb	UAF	✓	5
CVE-2016-20623	Binutils	1.0 Mb	UAF	✗	7

We also evaluate the implementation of AFLGoB and find it very close to the original AFLGo after accounting for emulation overhead.

UAF fuzzing benchmark The standard UAF micro benchmark Juliet Test Suite [NIS20] for static analyzers is too simple for fuzzing. No macro benchmark actually assesses the effectiveness of UAF detectors – the widely used LAVA [DGHK+16] only contains buffer overflows. Thus, we construct a new UAF benchmark according to the following rationale:

1. The subjects are real-world popular and fairly large security-critical programs;
2. The benchmark includes UAF bugs found by existing fuzzers [GZQ+18, CXL+18, BPR16, afl20a] or collected from National Vulnerability Database (NVD) [nvd20]. Especially, we include *all* UAF bugs found by directed fuzzers;
3. The bug report provides detailed information (e.g., buggy version and the stack trace), so that we can identify target locations for fuzzers.

In summary, we have 13 known UAF vulnerabilities (2 from directed fuzzers) over 11 real-world C programs whose sizes vary from 26 Kb to 3.8 Mb. Furthermore, selected programs range from image processing to data archiving, video processing and web development. Our benchmark is therefore representative of different UAF vulnerabilities of real-world programs. Table 4.3 presents our evaluation benchmark.

Evaluation configurations We follow the recommendations for fuzzing evaluations [KRC+18] and use the same fuzzing configurations and hardware resources for all experiments. Experiments are conducted 10 times with a time budget depending on the Program Under Test (PUT). We use as input seed either an empty file or existing valid files provided by developers. We do not use any token dictionary. All experiments were

carried out on an Intel Xeon CPU E3-1505M v6 @ 3.00GHz CPU with 32GB RAM and Ubuntu 16.04 64-bit.

4.4.3 UAF Bug-reproducing Ability (RQ1)

Protocol We compare the different fuzzers on our 13 UAF vulnerabilities using *Time-to-Exposure* (TTE), i.e. the time elapsed until first bug-triggering input, and *number of success runs* in which a fuzzer triggers the bug. In case a fuzzer cannot detect the bug within the time budget, the run's TTE is set to the time budget. Following existing work [BPNR17, CXL+18], we use the *Vargha-Delaney statistic* ($\hat{A}_{12}$) metric [VD00] [3] to assess the confidence that one tool outperforms another. Code coverage is not relevant for directed fuzzers.

Results Figure 4.6 presents a consolidated view of the results (total success runs and TTE – we denote by μTTE the average TTE observed for each sample over 10 runs). Table 4.4 summarizes the fuzzing performance (details in Table 4.5) of 4 binary-based fuzzers against our benchmark by providing the total number of covered paths, the total number of success runs and the max/min/average/median values of *Factor* and $\hat{A}_{12}$. Table 4.5 provides additional information: detailed statistics per benchmark sample.

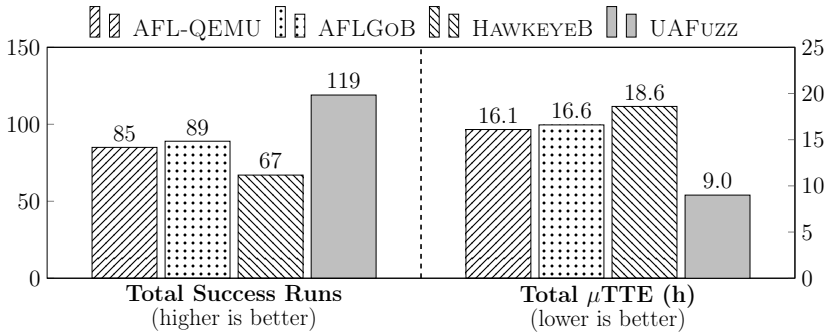


Figure 4.6: Summary of fuzzing performance (RQ1).

Table 4.4: Summary of bug reproduction of UAFuzz compared to other fuzzers against our fuzzing benchmark. Statistically significant results $\hat{A}_{12} \geq 0.71$ are marked as bold.

Fuzzer	Total Avg Paths	Success Runs	Factor				$\hat{A}_{12}$			
			Mdn	Avg	Min	Max	Mdn	Avg	Min	Max
AFL-QEMU	10.6K	85 (+40%)	2.01	6.66	0.60	46.63	**0.82**	**0.78**	0.29	**1.00**
AFLGoB	11.1K	89 (+34%)	1.96	6.73	0.96	43.34	**0.80**	**0.78**	0.52	**1.00**
HawkeyeB	7.3K	67 (+78%)	2.90	8.96	1.21	64.29	**0.88**	**0.86**	0.56	**1.00**
UAFuzz	8.2K	119	–	–	–	–	–	–	–	–

[3] Value between 0 and 1, the higher the better. Values above the conventionally large effect size of 0.71 are considered highly relevant [VD00].

Table 4.5: Bug reproduction on 4 fuzzers against our benchmark. Statistically significant results $\hat{A}_{12} \geq 0.71$ are marked as bold. *Factor* measures the performance gain as the μTTE of other fuzzers divided by the μTTE of UAFuzz.

Bug ID	Fuzzer	Paths	Runs	μTTE(s)	Factor	$\hat{A}_{12}$
giflib-bug-74	AFL-QEMU	196.0	10	290	1.39	0.64
	AFLGoB	172.9	9	478	2.29	0.70
	HawkeyeB	135.8	7	677	3.24	0.62
	UAFuzz	184.0	10	209	–	–
CVE-2018-11496	AFL-QEMU	404.0	10	19	1.36	**0.81**
	AFLGoB	339.8	10	22	1.57	**0.92**
	HawkeyeB	323.6	10	57	4.07	**1.00**
	UAFuzz	434.4	10	14	–	–
yasm-issue-91	AFL-QEMU	2110.0	8	2611	46.63	**1.00**
	AFLGoB	2018.3	8	2427	43.34	**1.00**
	HawkeyeB	323.2	0	3600	64.29	**1.00**
	UAFuzz	1364.1	10	56	–	–
CVE-2016-4487	AFL-QEMU	931.0	4	2661	1.26	0.62
	AFLGoB	1359.7	6	2427	1.15	0.57
	HawkeyeB	895.6	7	2559	1.21	0.56
	UAFuzz	1043.1	6	2110	–	–
CVE-2018-11416	AFL-QEMU	21.5	8	744	3.17	**0.96**
	AFLGoB	21.0	10	303	1.29	**0.78**
	HawkeyeB	21.0	10	338	1.44	**0.88**
	UAFuzz	21.0	10	235	–	–
mjs-issue-78	AFL-QEMU	1202.4	0	10800	2.57	**0.95**
	AFLGoB	1479.4	4	8755	2.09	**0.80**
	HawkeyeB	730.5	0	10800	2.57	**0.95**
	UAFuzz	867.9	9	4197	–	–
mjs-issue-73	AFL-QEMU	1462.5	1	9833	2.01	**0.82**
	AFLGoB	1314.3	0	10800	2.21	**0.85**
	HawkeyeB	741.6	0	10800	2.21	**0.85**
	UAFuzz	862.4	7	4881	–	–
CVE-2018-10685	AFL-QEMU	400.3	9	232	1.49	0.60
	AFLGoB	388.1	9	305	1.96	0.55
	HawkeyeB	316.6	5	500	3.21	**0.85**
	UAFuzz	352.7	10	156	–	–
CVE-2019-6455	AFL-QEMU	240.3	6	1149	2.62	**0.86**
	AFLGoB	206.0	5	1213	2.77	**0.81**
	HawkeyeB	205.7	5	1270	2.90	**0.86**
	UAFuzz	169.3	10	438	–	–
CVE-2017-10686	AFL-QEMU	2403.5	1	20905	2.08	**1.00**
	AFLGoB	2549.9	3	19721	1.96	**0.99**
	HawkeyeB	1937.4	1	20134	2.00	**0.99**
	UAFuzz	2190.3	10	10040	–	–
gifsicle-issue-122	AFL-QEMU	367.1	8	5938	0.60	0.29
	AFLGoB	383.4	6	9811	0.96	0.52
	HawkeyeB	256.4	4	12473	1.26	0.67
	UAFuzz	242.4	7	9853	–	–
CVE-2016-3189	AFL-QEMU	117.0	10	149	1.06	0.59
	AFLGoB	125.1	10	158	1.12	0.66
	HawkeyeB	67.4	10	770	5.46	**1.00**
	UAFuzz	100.1	10	141	–	–
CVE-2018-20623	AFL-QEMU	804.0	10	2604	20.34	**1.00**
	AFLGoB	724.2	9	3169	24.76	**1.00**
	HawkeyeB	625.1	8	2889	22.57	**1.00**
	UAFuzz	388.6	10	128	–	–

Figure 4.6 (and Tables 4.4 and 4.5) show that UAFuzz clearly outperforms the other fuzzers both in total success runs (vs. 2nd best AFLGoB: +34% in total, up to +300%) and in TTE (vs. 2nd best AFLGoB, total: 2.0×, avg: 6.7×, max: 43×). In some specific cases (see Table 4.5), UAFuzz saves roughly 10,000s of TTE over AFLGoB or goes from 0/10 successes to 7/10. The $\hat{A}_{12}$ value of UAFuzz against other fuzzers is also significantly above the conventional large effect size 0.71 [VD00], as shown in Table 4.4 (vs. 2nd best AFLGoB, avg: 0.78, median: 0.80, min: 0.52). Figure 4.7 finally shows UAFuzz to have more stable performance.

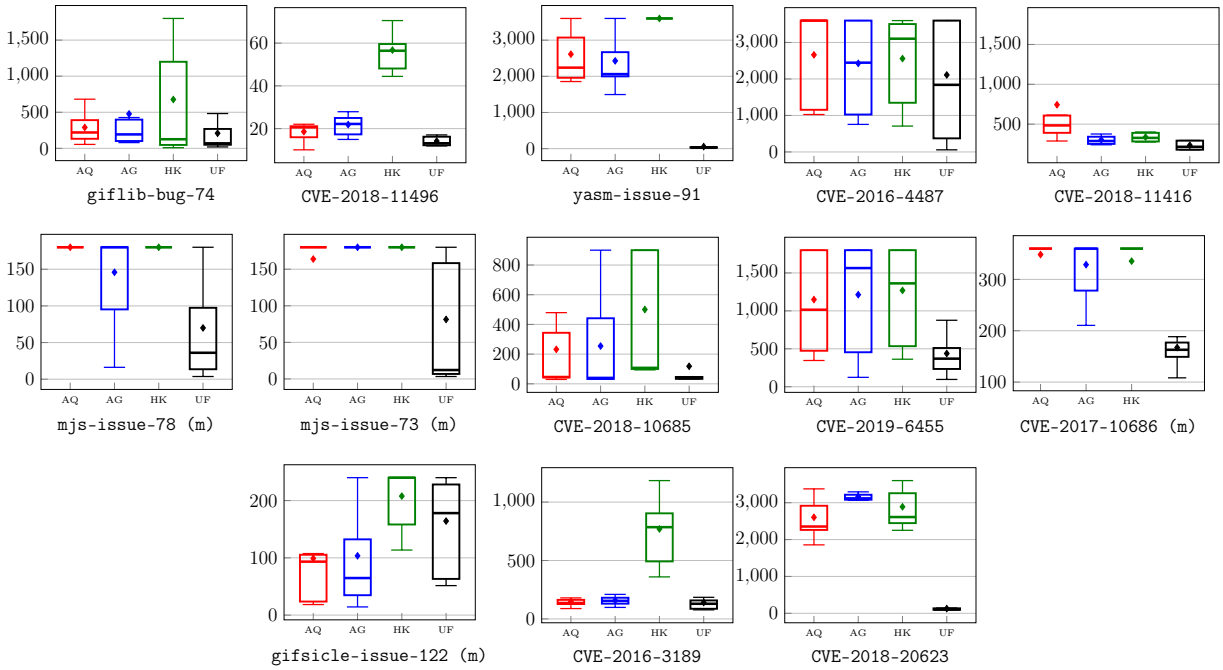


Figure 4.7: TTE in seconds of 4 fuzzers except for subjects marked with "(m)" for which the unit is minute (lower is better). AQ, AG, HK and UF denote AFL-QEMU, AFLGoB, HawkeyeB and UAFuzz, respectively.

Answer to RQ1: UAFuzz *significantly* outperforms state-of-the-art directed fuzzers in terms of UAF bugs reproduction with a high confidence level.

Zoom on `yasm-issue-91` We discuss the case of `yasm-issue-91`, where in all 10 runs, UAFuzz needs only in a few seconds to reproduce the bug, thus gains a speedup of 43× over the second best tool AFLGoB with a high confidence (i.e., $\hat{A}_{12}$ is 1 against other fuzzers). Figure 4.8 depicts the fuzzing queue of our fuzzer UAFuzz for the case study in one run. We can see that our seed selection heuristic first selects the most promising inputs among the set of initial test suite (i.e., the most left circle point). As this input also has the biggest cut-edge score among the initial seeds, UAFuzz spends enough long time to mutate this input and thus eventually discovers the first potential input whose execution trace is similar to the expected trace. Then, two first potential inputs covering in sequence all 19 targets are selected to be mutated by UAFuzz during fuzzing. Consequently, UAFuzz could trigger the bug at the third potential input (i.e., the 954th input in the fuzzing queue). Overall in 10 runs the first bug-triggering input of UAFuzz is the 1019th on average, while for AFL-QEMU and AFLGoB they detect the bug much slower, at the 2026th and 1908th input respectively. The main reason is that other tools spend more time on increasing the code coverage by going through all initial seeds in the fuzzing queue. In particular, as AFLGoB aims to first explore more paths in the exploration phase, it is more likely that

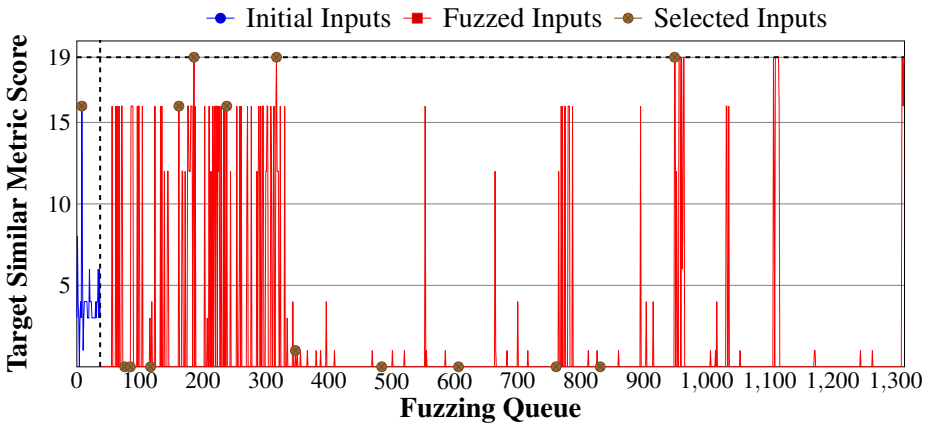


Figure 4.8: Fuzzing queue of UAFuzz for `yasm-issue-91`. Selected inputs to be mutated are highlighted in brown. Potential inputs are in the horizontal dashed line.

directed fuzzers that are mainly based on the seed distance metric like AFLGoB skip or select the input after long time. Although both AFL-QEMU and AFLGoB could find the bug in 8 and 10 runs and discover substantially more paths than our fuzzer, the TTE values of these tools are clearly much more larger than UAFuzz's TTE.

Comparison between AFLGoB **and source-based** AFLGo We want to evaluate how close our implementation of AFLGoB is from the original AFLGo, in order to assess the degree of confidence we can have in our results – we do not do it for HawkeyeB as Hawkeye is not available.

AFLGo unsurprisingly performs better than AFLGoB and UAFuzz (Figure 4.9, Table 4.6). This is largely due to the emulation runtime overhead of QEMU, a well-documented fact. Still, *surprisingly enough*, UAFuzz can find the bugs faster than AFLGo in 4 samples, demonstrating its efficiency.

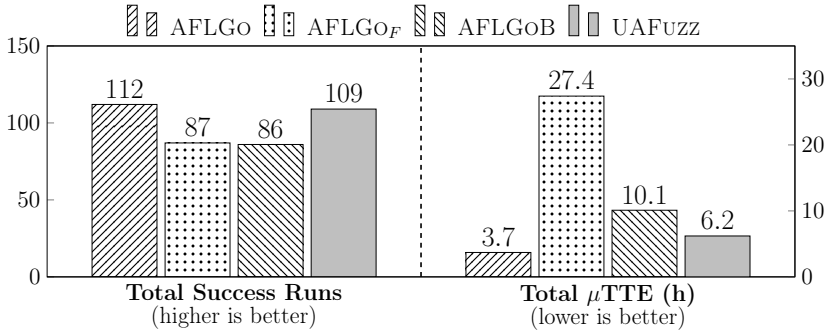


Figure 4.9: Summary of fuzzing performance of 4 fuzzers against our benchmark, except `CVE-2017-10686` due to compilation issues of AFLGo.

Table 4.6: Bug reproduction of AFLGo against our benchmark except `CVE-2017-10686` due to compilation issues of AFLGo. Numbers in red are the best μTTEs.

Bug ID	AFLGo (source)		AFLGo_F (source)		AFLGoB		UAFuzz	
	Runs	μTTE(s)	Runs	μTTE(s)	Runs	μTTE(s)	Runs	μTTE(s)
`giflib-bug-74`	10	62	10	281	9	478	10	209
`CVE-2018-11496`	10	2	10	38	10	22	10	14
`yasm-issue-91`	10	307	8	2935	8	2427	10	56
`CVE-2016-4487`	10	676	10	1386	6	2427	6	2110
`CVE-2018-11416`	10	78	7	1219	10	303	10	235
`mjs-issue-78`	10	1417	3	9706	4	8755	9	4197
`mjs-issue-73`	9	5207	3	34210	0	10800	7	4881
`CVE-2018-10685`	10	74	9	1072	9	305	10	156
`CVE-2019-6455`	5	1090	0	20296	5	1213	10	438
`gifsicle-issue-122`	8	4161	7	25881	6	9811	7	9853
`CVE-2016-3189`	10	72	10	206	10	158	10	141
`CVE-2018-20623`	10	177	10	1329	9	3169	10	128
Total Success Runs	**112**		**87**		**86**		**109**	
Total μTTE (h)	**3.7**		**27.4**		**10.1**		**6.2**	

Yet, more interestingly, Figure 4.9 also shows that once emulation overhead [4] is taken into account (yielding AFLGo_F, the expected *binary-level* performance of AFLGo), then AFLGoB is in line with AFLGo_F (and even shows better TTE) – UAFuzz even significantly outperforms AFLGo_F.

Answer to RQ1: Performance of AFLGoB is in line with the original AFLGo once QEMU overhead is taken into account, allowing a fair comparison with UAFuzz. UAFuzz nonetheless performs relatively well on UAF compared with the source-based directed fuzzer AFLGo, demonstrating the benefit of our original fuzzing mechanisms.

4.4.4 UAF Overhead (RQ2)

Protocol We are interested in both (1) instrumentation-time overhead and (2) runtime overhead. For (1), we simply compute the total instrumentation time of UAFuzz and we compare it to the instrumentation time of AFLGo. For (2), we compute the total number of executions per second of UAFuzz and compare it AFL-QEMU taken as a baseline.

[4] We estimate for each sample an overhead factor f by comparing the number of executions per second in both AFL and AFL-QEMU, then multiply the computation time of AFLGo by f – f varies from 2.05 to 22.5 in our samples.

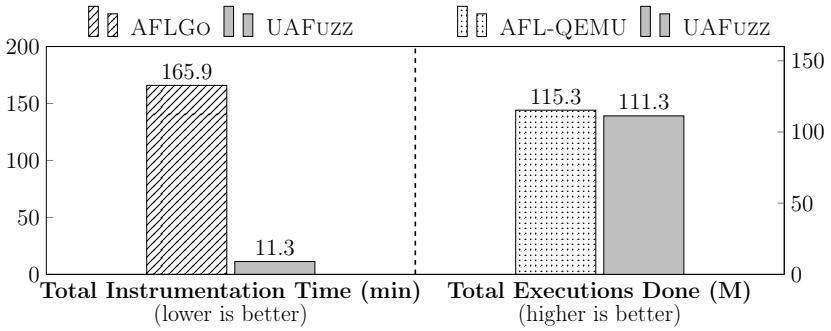


Figure 4.10: Global overhead (RQ2).

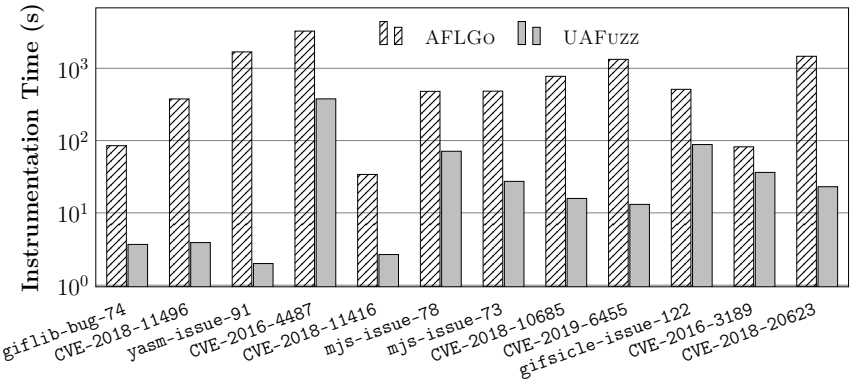


Figure 4.11: Average instrumentation time in seconds (except `CVE-2017-10686` due to compilation issues of AFLGo).

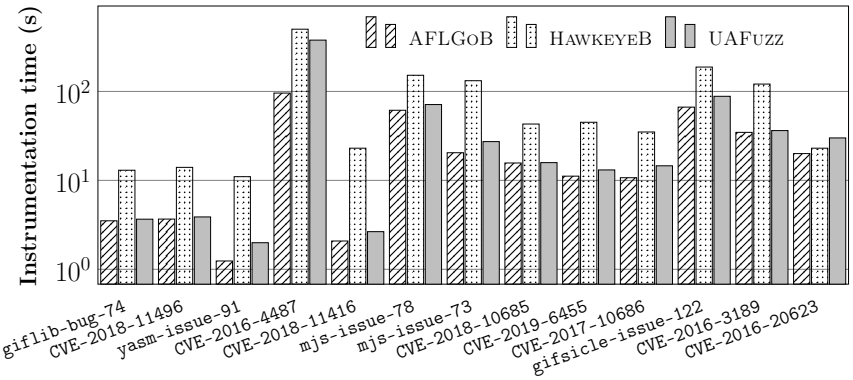


Figure 4.12: Average instrumentation time in seconds.

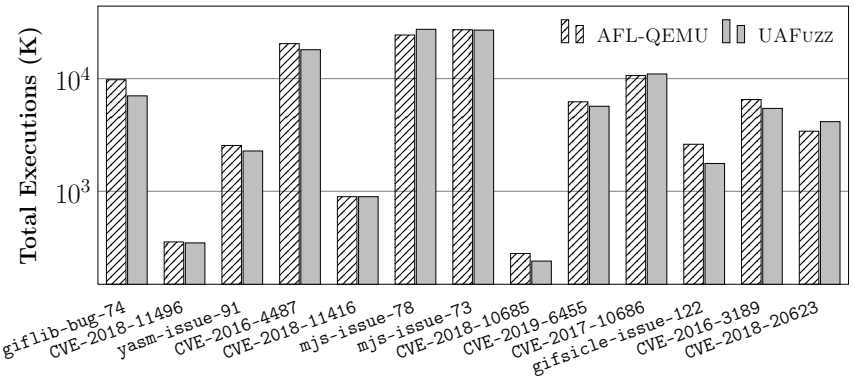


Figure 4.13: Total executions done in all runs.

Results Consolidated results for both instrumentation-time and runtime overhead are presented in Figure 4.10 (number of executions per second is replaced by the total number of executions performed in the same time budget). This figure shows that UAFuzz is *an order of magnitude faster than the state-of-the-art source-based directed fuzzer* AFLGo *in the instrumentation phase*, and has almost the same total number of executions per second as AFL-QEMU.

We also provide additional results for RQ2. Figures 4.11 and 4.12 compare the average instrumentation time between, respectively, UAFuzz and the source-based directed fuzzer AFLGo; and UAFuzz and the two binary-based directed fuzzers AFLGoB and HawkeyeB. Figure 4.13 shows the total execution done of AFL-QEMU and UAFuzz for each subject in our benchmark. Figure 4.14 compares the average triaging time between UAFuzz and other fuzzers against our benchmark. We now discuss experimental results regarding overhead in more depth as follows:

- Figures 4.10 and 4.11 show that UAFuzz is *an order of magnitude faster than the state-of-the-art source-based directed fuzzer* AFLGo *in the instrumentation phase* (14.7× faster in total). For example, UAFuzz spends only 23s (i.e., 64× less than AFLGo) in processing the large program `readelf` of Binutils;

- Figures 4.10 and 4.13 show that UAFuzz has almost the same total number of executions per second as AFL-QEMU (-4% in total, -12% in average), meaning that its overhead is negligible.

- Figure 4.12 shows that HawkeyeB is sometimes significantly slower than UAFuzz (2×). This is mainly because of the cost of target function trace closure calculation on large examples with many functions.

Answer to RQ2: UAFuzz enjoys both a *lightweight instrumentation time* and a *minimal runtime overhead.*

Table 4.7: Average number of triaging inputs of 4 fuzzers against our tested subjects. For UAFuzz, the TIR values are in parentheses.

Bug ID	AFL-QEMU	AFLGoB	HawkeyeB	UAFuzz
giflib-bug-74	200.9	177.0	139.9	10.0 (5.31%)
CVE-2018-11496	409.6	351.7	332.5	5.4 (4.08%)
yasm-issue-91	2115.3	2023.0	326.6	37.4 (2.72%)
CVE-2016-4487	933.1	1367.2	900.2	2.5 (0.24%)
CVE-2018-11416	21.5	21.0	21.0	1.0 (4.76%)
mjs-issue-78	1226.9	1537.8	734.6	262.3 (30.22%)
mjs-issue-73	1505.6	1375.9	745.6	252.2 (29.25%)
CVE-2018-10685	414.2	402.1	328.9	12.6 (3.14%)
CVE-2019-6455	243.2	238.1	211.1	6.9 (1.57%)
CVE-2017-10686	2416.9	2517.0	1765.2	214.3 (8.96%)
gifsicle-issue-122	405.0	431.7	378.5	3.3 (0.86%)
CVE-2016-3189	377.9	764.7	126.4	7.1 (1.69%)
CVE-2018-20623	804.0	724.2	625.1	5.4 (1.39 %)
Total	**11.1K**	**11.9K**	**6.6K**	**820 (7.25%)**

4.4.5 UAF Triage (RQ3)

Protocol We consider the total number of triaging inputs (number of inputs sent to the triaging step), the triaging inputs rate TIR (ratio between the total number of generated inputs and those sent to triaging) and the total triaging time (time spent within the triaging step). Since other fuzzers cannot identify inputs reaching targets during the fuzzing process, we conservatively analyze *all* inputs generated by the these fuzzers in the bug triage step (TIR = 1).

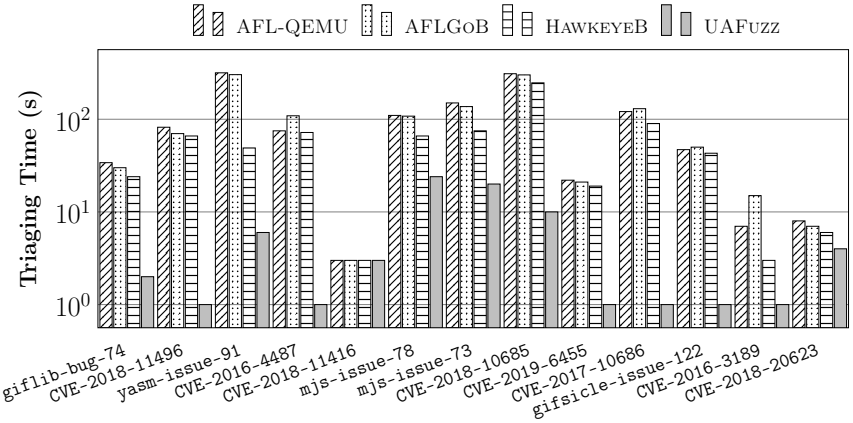


Figure 4.14: Average triaging time in seconds.

Results Consolidated results are presented in Figure 4.15, detailed results in Table 4.7 and Figure 4.14.

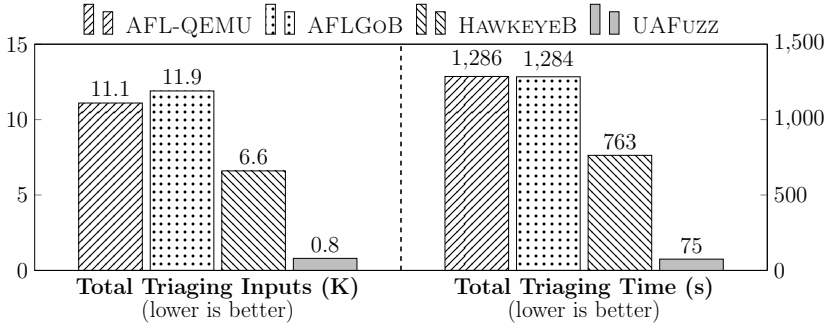


Figure 4.15: Summary of bugs triage (RQ3).

- The TIR of UAFuzz is 9.2% in total (avg: 7.25%, median: 3.14%, best: 0.24%, worst: 30.22%) – sparing up to 99.76% of input seeds for confirmation, and is always less than 9% except for sample `mjs`;

- Figure 4.14 shows that UAFuzz spends the smallest amount of time in bug triage, i.e. 75s (avg: 6s, min: 1s, max: 24s) for a total speedup of 17× over AFLGoB (max: 130×, avg: 39×).

Answer to RQ3: UAFuzz reduces a *large* portion (i.e., more than 90%) of triaging inputs in the post-processing phase. Subsequently, UAFuzz only spends several seconds in this step, winning an order of magnitude compared to standard directed fuzzers.

4.4.6 Individual Contribution (RQ4)

Protocol We compare four different versions of our prototype, representing a continuum between AFLGo and UAFuzz: (1) the basic AFLGo represented by AFLGoB, (2) AFLGoB–ss adds our seed selection metric to AFLGoB, (3) AFLGoB–ds adds the UAF-based function distance to AFLGoB–ss, and finally (4) UAFuzz adds our dedicated power schedule to AFLGoB–ds. We consider the previous RQ1 metrics: number of success runs, TTE and Vargha-Delaney. Our goal is to assess whether or not these technical improvements do lead to fuzzing performance improvements.

Results Consolidated results for success runs and TTE are represented in Figure 4.16. As summarized in Figure 4.16, we can observe that each new component does improve both TTE and number of success runs, leading indeed to fuzzing improvement. Detailed results in Table 4.8 with $\hat{A}_{12}$ values show the same clear trend.

Table 4.8: Bug reproduction on 4 fuzzers against our benchmark. $\hat{A}_{12A}$ and $\hat{A}_{12U}$ denote the Vargha-Delaney values of AFLGoB and UAFuzz. Statistically significant results for $\hat{A}_{12}$ (e.g., $\hat{A}_{12A} \leq 0.29$ or $\hat{A}_{12U} \geq 0.71$) are in bold. Numbers in red are the best μTTEs.

Bug ID	AFLGoB			AFLGoB–ss				AFLGoB–ds				UAFuzz		
	Runs	μTTE(s)	$\hat{A}_{12U}$	Runs	μTTE(s)	$\hat{A}_{12A}$	$\hat{A}_{12U}$	Runs	μTTE(s)	$\hat{A}_{12A}$	$\hat{A}_{12U}$	Runs	μTTE(s)	$\hat{A}_{12A}$
giflib-bug-74	9	478	0.70	10	261	0.47	0.66	10	317	0.47	0.67	10	209	0.30
CVE-2018-11496	10	22	**0.92**	10	14	**0.06**	0.44	10	23	0.52	**1.00**	10	14	**0.08**
yasm-issue-91	8	2427	**1.00**	10	37	**0.00**	0.44	10	99	**0.00**	0.47	10	56	**0.00**
CVE-2016-4487	6	2427	0.57	5	2206	0.46	0.53	5	2494	0.51	0.59	6	2110	0.43
CVE-2018-11416	10	303	**0.78**	10	232	**0.24**	0.50	10	408	0.79	**0.88**	10	235	**0.22**
mjs-issue-78	4	8755	**0.80**	4	7454	0.47	**0.72**	9	3707	**0.22**	0.48	9	4197	**0.20**
mjs-issue-73	0	10800	**0.85**	3	7651	0.35	0.68	6	5432	**0.20**	0.56	7	4881	**0.15**
CVE-2018-10685	9	305	0.57	10	128	0.43	0.47	10	160	0.54	0.67	10	118	0.43
CVE-2019-6455	5	1213	**0.81**	10	407	**0.19**	0.48	9	981	0.37	**0.75**	10	438	**0.19**
CVE-2017-10686	3	19721	**0.99**	10	12838	**0.07**	**0.73**	10	12484	**0.07**	0.69	10	10040	**0.01**
gifsicle-issue-122	8	6210	0.52	3	12702	0.68	0.72	2	13443	0.72	**0.77**	7	9853	0.48
CVE-2016-3189	10	158	0.66	10	141	0.35	0.55	10	152	0.40	0.55	10	141	0.34
CVE-2018-20623	9	3169	**1.00**	10	135	**0.00**	0.10	10	89	**0.00**	0.18	10	128	**0.00**
Total Success Runs	**89**			**105 (+18.0%)**				**111 (+24.7%)**				**119 (+33.7%)**		
Total μTTE (h)	**15.6**			**12.3**				**11.1**				**9.0**		
Average $\hat{A}_{12A}$	–			**0.29**				0.37				**0.22**		
Average $\hat{A}_{12U}$	**0.78**			0.54				0.64				–		

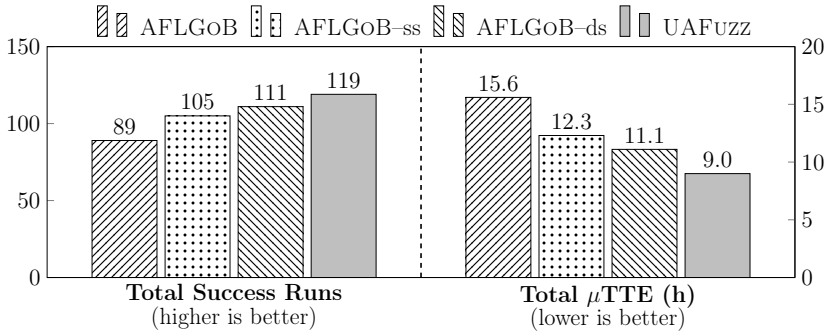


Figure 4.16: Impact of each components (RQ4).

Answer to RQ4: The UAF-based distance computation, the power scheduling and the seed selection heuristic *individually* contribute to improve fuzzing performance, and combining them yield even further improvements, demonstrating their interest and complementarity.

4.4.7 Patch Testing & Zero-days

Patch testing The idea is to use bug stack traces of *known* UAF bugs to guide testing on the *patched* version of the PUT – instead of the buggy version as in bug reproduction. The benefit from the bug hunting point of view [gpz20] is both to try finding buggy or incomplete patches *and* to focus testing on *a priori* fragile parts of the code, possibly discovering bugs unrelated to the patch itself.

How to We follow bug hunting practice [gpz20]. Starting from the recent publicly disclosed UAF bugs of open source programs, we manually identify addresses of relevant call instructions in the reported bug stack traces since the code has been evolved. We focus mainly on 3 widely-used programs that have been well fuzzed and maintained by the developers, namely GNU patch, GPAC and Perl 5 (737K lines of C code and 5 known bug traces in total). We also consider 3 other codes: MuPDF, Boolector and fontforge (+1,196Kloc).

Table 4.9: Summary of zero-day vulnerabilities reported by our fuzzer UAFuzz (**32** new bugs including **13** new UAF bugs, **10** CVEs were assigned and **23** bugs were fixed).

Program	Code Size	Version (Commit)	Bug ID	Vulnerability Type	Crash	Vulnerable Function	Status	CVE
GPAC	545K	0.7.1 (987169b)	#1269	**User after free**	✗	gf_m2ts_process_pmt	Fixed	CVE-2019-20628
		0.8.0 (56eaea8)	#1440-1	**User after free**	✗	gf_isom_box_del	Fixed	CVE-2020-11558
		0.8.0 (56eaea8)	#1440-2	**User after free**	✗	gf_isom_box_del	Fixed	
		0.8.0 (56eaea8)	#1440-3	**User after free**	✗	gf_isom_box_del	Fixed	
		0.8.0 (5b37b21)	#1427	**User after free**	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1263	NULL pointer dereference	✓	ilst_item_Read	Fixed	
		0.7.1 (987169b)	#1264	Heap buffer overflow	✓	gf_m2ts_process_pmt	Fixed	CVE-2019-20629
		0.7.1 (987169b)	#1265	Invalid read	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1266	Invalid read	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1267	NULL pointer dereference	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1268	Heap buffer overflow	✓	BS_ReadByte	Fixed	CVE-2019-20630
		0.7.1 (987169b)	#1270	Invalid read	✓	gf_list_count	Fixed	CVE-2019-20631
		0.7.1 (987169b)	#1271	Invalid read	✓	gf_odf_delete_descriptor	Fixed	CVE-2019-20632
		0.8.0 (5b37b21)	#1445	Heap buffer overflow	✓	gf_bs_read_data	Fixed	
		0.8.0 (5b37b21)	#1446	Stack buffer overflow	✓	gf_m2ts_get_adaptation_field	Fixed	
GNU patch	7K	2.7.6 (76e7758)	#56683	**Double free**	✓	another_hunk	Confirmed	CVE-2019-20633
		2.7.6 (76e7758)	#56681	Assertion failure	✓	pch_swap	Confirmed	
		2.7.6 (76e7758)	#56684	Memory leak	✗	xmalloc	Confirmed	
MuPDF	539K	1.16.1 (6566de7)	#702253	**Use after free**	✗	fz_drop_band_writer	Fixed	CVE-2020-16600
Perl 5	184K	5.31.3 (a3c7756)	#134324	**Use after free**	✓	S_reg	Confirmed	
		5.31.3 (a3c7756)	#134326	**Use after free**	✓	Perl_regnext	Fixed	
		5.31.3 (a3c7756)	#134329	**User after free**	✓	Perl_regnext	Fixed	
		5.31.3 (a3c7756)	#134322	NULL pointer dereference	✓	do_clean_named_objs	Confirmed	
		5.31.3 (a3c7756)	#134325	Heap buffer overflow	✓	S_reg	Fixed	
		5.31.3 (a3c7756)	#134327	Invalid read	✓	S_regmatch	Fixed	
		5.31.3 (a3c7756)	#134328	Invalid read	✓	S_regmatch	Fixed	
		5.31.3 (45f8e7b)	#134342	Invalid read	✓	Perl_mro_isa_changed_in	Confirmed	
Boolector	79K	3.2.1 (3249ae0)	#90	NULL pointer dereference	✓	set_last_occurrence_of_symbols	Won't fix	
fontforge	578K	20200314 (1604c74)	#4266	**Use after free**	✓	SFDGetBitmapChar	Won't fix	
		20200314 (1604c74)	#4267	NULL pointer dereference	✓	SFDGetBitmapChar	Won't fix	
readelf	1.0 M	2.34 (f717994)	#25821	**Double free**	✓	process_symbol_table	Fixed	CVE-2020-16590
nm-new	6.7 M	2.34 (c98a454)	#25823	**Use after free**	✓	bfd_hash_lookup	Fixed	CVE-2020-16592

Results Overall UAFuzz has found and reported **32 new bugs**, including **13 new UAF bugs** and **10 new CVEs** (details in Table 4.9). *At this time, 23 bugs have been fixed by the vendors.* Interestingly, the bugs found in GNU patch and GPAC were actually *buggy patches*.

Zoom: GNU Patch buggy patch We use `CVE-2018-6952` [cve20b] to demonstrate the effectiveness of UAFuzz in exposing unknown UAF vulnerabilities. GNU patch [gnu20] takes a patch file containing a list of differences and applies them to the original file. Listing 4.2 shows the code fragment of `CVE-2018-6952` which is a double free in the latest version 2.7.6 of GNU patch. Interestingly, by using the stack trace of this CVE as shown in Figure 4.17, UAFuzz successfully discovered an *incomplete bug fix* [dfp20] in the latest commit `76e7758`, with a slight difference of the bug stack trace (i.e., the call of `savebuf()` in `another_hunk()`).

```
1 File:  src/patch.c
2 int main (int argc, char **argv) {...
3   while (0 < (got_hunk = another_hunk (diff_type, reverse))) {
4   /* Apply each hunk of patch */ ... }
5 ...}
6
7 File:  src/pch.c
8 int another_hunk (enum diff difftype, bool rev) { ...
9   while (p_end >= 0) {
10    if (p_end == p_efake) p_end = p_bfake;
11    else free(p_line[p_end]); /* Free and Use event */
12    p_end--;
13  } ...
14  while (p_end < p_max) { ...
15    switch(*buf) { ...
16    case '+': case '!': /* Our bug CVE-2019-20633 */ ...
17      p_line[p_end] = savebuf (s, chars_read); ...
18    case ' ': /* CVE-2018-6952 */ ...
19      p_line[p_end] = savebuf (s, chars_read); ...
20    ...}
21  ...}
22 ... }
23
24 File:  src/util.c
25 /* Allocate a unique area for a string. */
26 char *savebuf (char const *s, size_t size) { ...
27  rv = malloc (size); /* Alloc event */ ...
28  memcpy (rv, s, size);
29  return rv;
30 }
```

Listing 4.2: Code fragment of GNU patch pertaining to the UAF vulnerability `CVE-2018-6952`.

Technically, GNU patch takes an input patch file containing multiple hunks (line 4) that are split into multiple strings using special characters as delimiter via `*buf` in the switch case (line 15). GNU patch then reads and parses each string stored in `p_line` that is dynamically allocated on the memory using `malloc()` in `savebuf()` (line 27) until the last line of this hunk has been processed. Otherwise, GNU patch deallocates the most recently processed string using `free()` (line 11). Our reported bug and `CVE-2018-6952` share the same *free* and *use* event, but have a different stack trace leading to the same *alloc* event. Actually, while the PoC input generated by UAFuzz contains two characters '`!`', the PoC of `CVE-2018-6952` does not contain this character, consequently the case in line 17 was previously uncovered, and thus this CVE had been incompletely fixed. This case study shows the importance of producing different unique bug-triggering inputs to favor the repair process and help complete bug fixing.

```
    // Stack trace for the bad Use (here:  a free)
==330== Invalid free() / delete / delete[] / realloc()
==330== at 0x402D358:  free (in vgpreload_memcheck-x86-linux.so)
==330== by 0x8052E11:  another_hunk (pch.c:1185)
==330== by 0x804C06C: main (patch.c:396)

    // Stack trace for the Free
==330== Address 0x4283540 is 0 bytes inside a block of size 2 free'd
==330== at 0x402D358:  free (in vgpreload_memcheck-x86-linux.so)
==330== by 0x8052E11:  another_hunk (pch.c:1185)
==330== by 0x804C06C: main (patch.c:396)

    // Stack trace for the Alloc
==330== Block was alloc'd at
==330== at 0x402C17C: malloc (in vgpreload_memcheck-x86-linux.so)
==330== by 0x805A821:  savebuf (util.c:861)
==330== by 0x805423C: another_hunk (pch.c:1504)
==330== by 0x804C06C: main (patch.c:396)
```

Figure 4.17: The bug trace of CVE-2018-6952 (Double Free) produced by VALGRIND.

UAFUZZ has been proven effective in a patch testing setting, allowing to find 32 new bugs (incl. 10 new CVEs) in 8 widely-used programs.

4.4.8 Threats to Validity

Implementation Our prototype is implemented as part of the binary-level code analysis framework BINSEC [DBT+16, DB15], whose efficiency and robustness have been demonstrated in prior large scale studies on both adversarial code and managed code [BDM17, RBB+19, DBF+16], and on top of the popular fuzzer AFL-QEMU. Effectiveness and correctness of UAFUZZ have been assessed on several bug traces from real programs, as well as on small samples from the Juliet Test Suite. All reported UAF bugs have been manually checked.

Benchmark Our benchmark is built on both real codes *and* real bugs, and encompass several bugs found by recent fuzzing techniques of well-known open source codes (including all UAF bugs found by directed fuzzers).

Competitors We consider the best state-of-the-art techniques in directed fuzzing, namely AFLGO [BPNR17] and HAWKEYE [CXL+18]. Unfortunately, HAWKEYE is not available and AFLGO works on source code only. Thus, we re-implement these technologies in our own framework. We followed the available information (article, source code if any) as close as possible, and did our best to get precise implementations. They have both

been checked on real programs and small samples, and the comparison against AFLGo source and our own AFLGoB implementation is conclusive.

4.5 Related Work

4.5.1 Directed Greybox Fuzzing

The state-of-the-art directed fuzzers such as AFLGo [BPNR17] and Hawkeye [CXL+18] have already been discussed. LOLLY [LZY+19] provides a lightweight instrumentation to measure the sequence basic block coverage of inputs, yet, at the price of a large runtime overhead. SeededFuzz [WSZ16] seeks to generate a set of initial seeds that improves directed fuzzing performance. Our fuzzer UAFuzz could therefore benefit from the improved seed selection and generation techniques of SeededFuzz. SemFuzz [YZC+17] leverages vulnerability-related texts such as CVE reports to guide fuzzing and automatically generate PoC exploits for Linux kernel flaws. 1dVul [PLL+19] discovers 1-day vulnerabilities via binary patches by leveraging a hybrid approach of distance-based directed fuzzing and dominator-based directed symbolic execution. Different from these works, we use UAF bug traces to guide the fuzzing to detect specific UAF bugs in software binaries.

UAFuzz *is the first directed fuzzer tailored to UAF bugs, and one of the very few [PLL+19] able to handle binary code.*

4.5.2 Coverage-based Greybox Fuzzing

AFL [afl20a] is the seminal coverage-guided greybox fuzzer. Substantial efforts have been conducted in the last few years to improve over it [BPR16, LS18, GZQ+18]. Also, many efforts have been fruitfully invested in combining fuzzing with other approaches, such as static analysis [LCC+17, GZQ+18], dynamic taint analysis [RJK+17, CC18, CLC19], symbolic execution [SGS+16, PSP18, YLX+18] or machine learning [GPS17, SPE+18].

Recently, UAFL [WXL+20] – another independent research effort on the same problem, specialized coverage-guided fuzzing to detect UAFs by finding operation sequences potentially violating a typestate property and then guiding the fuzzing process to trigger property violations. However, this approach relies heavily on the static analysis of source code, therefore is not applicable at binary-level.

Our technique is orthogonal to all these improvements, they could be reused within UAFuzz *as is.*

4.5.3 UAF Detection

Precise static UAF detection is difficult. GUEB [gue20] is the only binary-level static analyzer for UAF. The technique can be combined with dynamic symbolic execution to generate PoC inputs [FMB+16], yet with scalability issues. On the other hand, several UAF source-level static detectors exist, based on abstract interpretation [CKK+12], pointer

analysis [YSCX18], pattern matching [OHLP14], model checking [KT14] or demand-driven pointer analysis [SX16]. A common weakness of all static detectors is their inability to infer triggering input – they rather prove their absence.

Dynamic UAF detectors mainly rely on heavyweight instrumentation [CGMN12, NS07, drm20] and result in high runtime overhead, even more for closed source programs. ASan [SBPV12] performs lightweight instrumentation, but at source level only.

4.5.4 UAF Fuzzing Benchmark

Table 4.10: Summary of existing benchmarks.

~: DARPA CGC features crafted codes and bugs, yet they are supposed to be realistic

Benchmark	Realistic		#Programs	#Bugs	#UAF
	Program	Bug			
Juliet Test Suite [NIS20]	✗	✗	366	366	366
LAVA-1 [DGHK+16]	✓	✗	1	69	0
LAVA-M [DGHK+16]	✓	✗	4	2265	0
Apocalypse [RPDGH18]	✓	✗	30	30	0
Rode0day [rod20, FLDGB19]	✓	✗	44	2103	0
Google Fuzzer TestSuite [gft20]	✓	✓	24	46	3
FuzzBench [fuz20]	✓	✓	23	23	0
DARPA CGC [cgc20]	~	~	296	248	10
UAFuzz Benchmark (evaluation) [uaf20a]	✓	✓	**11**	**13**	**13**
UAFuzz Benchmark (full) [uaf20a]	✓	✓	**17**	**30**	**30**

While the Juliet Test Suite [NIS20] (CWE-415, CWE-416)[5] contains only too small programs, popular fuzzing benchmarks [DGHK+16, RPDGH18, rod20, gft20, cgc20] comprise only very few UAF bugs. Moreover, many of these benchmarks contain either artificial bugs [DGHK+16, RPDGH18, rod20, cgc20] or artificial programs [NIS20]. Recently, a ground-truth fuzzing benchmark Magma [HHP20], that contains real bugs in real software, allows to uniform fuzzer evaluation and comparison. Table 4.10 compares our UAF Fuzzing benchmarks to existing fuzzing benchmarks. Table 4.11 provides additional details about our evaluation benchmark, including program executables under test, buggy commits and fuzzing configurations (test driver, seeds and timeout).

Merging our evaluation benchmark (known UAF) and our new UAF bugs, we provide the largest fuzzing benchmark dedicated to UAF – 17 real codes and 30 real bugs [uaf20a].

[5] Juliet is mostly used for the evaluation of C/C++ static analysis tools.

Table 4.11: Detailed view of our evaluation benchmark.

Bug ID	Program			Bug			Fuzzing Configuration			
	Project	Size	Commit	Type	Crash	Found by	Test driver	Seeds	Timeout	# Targets
giflib-bug-74	GIFLIB	59 Kb	72e31ff	DF	✗	–	gifsponge <@@	"GIF"	30m	7
CVE-2018-11496	lrzip	581 Kb	ed51e14	UAF	✗	–	lrzip -t @@	lrz files	15m	12
yasm-issue-91	yasm	1.4 Mb	6caf151	UAF	✗	AFL	yasm @@	asm files	1h	19
CVE-2016-4487	Binutils	3.8 Mb	2c49145	UAF	✓	AFLFast	cxxfilt <@@	empty file	1h	7
CVE-2018-11416	jpegoptim	62 Kb	d23abf2	DF	✗	–	jpegoptim @@	jpeg files	30m	5
mjs-issue-78	mjs	255 Kb	9eae0e6	UAF	✗	Hawkeye	mjs -f @@	js files	3h	19
mjs-issue-73	mjs	254 Kb	e4ea33a	UAF	✗	Hawkeye	mjs -f @@	js files	3h	28
CVE-2018-10685	lrzip	576 Kb	9de7ccb	UAF	✗	AFL	lrzip -t @@	lrz files	15m	7
CVE-2019-6455	Recutils	604 Kb	97d20cc	DF	✗	–	rec2csv @@	empty file	30m	15
CVE-2017-10686	NASM	1.8 Mb	7a81ead	UAF	✓	CollAFL	nasm -f bin @@ -o /dev/null	asm files	6h	10
gifsicle-issue-122	Gifsicle	374 Kb	fad477c	DF	✗	Eclipser	gifsicle @@ test.gif -o /dev/null	"GIF"	4h	11
CVE-2016-3189	bzip2	26 Kb	962d606	UAF	✓	–	bzip2recover @@	bz2 files	30m	5
CVE-2018-20623	Binutils	1.0 Mb	923c6a7	UAF	✗	AFL	readelf -a @@	binary files	1h	7

4.6 Conclusion

UAFuzz is the *first directed* greybox fuzzing approach tailored to detecting UAF vulnerabilities (in binary) given only the bug stack trace. UAFuzz outperforms existing directed fuzzers, both in terms of time to bug exposure and number of successful runs. UAFuzz has been proven effective in both bug reproduction and patch testing.

Chapter 5

Implementation

Contents

5.1 Introduction **65**
5.2 Preprocessing **67**
5.2.1 Bug trace generation 67
5.2.2 BinIda Plugin 68
5.3 Core Fuzzing Engine **70**
5.3.1 Debugging with afl-showmap 70
5.3.2 Overhead 70
5.4 Examples **71**
5.4.1 Application 1: Bug Reproduction 71
5.4.2 Application 2: Patch Testing 75
5.5 Conclusion **76**

This chapter introduces the detailed implementation of our fuzzer UAFuzz. First, we discuss an overview of the workflow. Then, we go through each principle component of the UAFuzz's workflow, namely the preprocessing step, the BinIda plugin and the guided fuzzing system built on top of AFL-QEMU. Finally, we provide detailed instructions to run UAFuzz in two scenarios: bug reproduction and patch testing.

5.1 Introduction

Workflow We have implemented our results in a UAF-oriented binary-level directed fuzzer, named UAFuzz. Figure 5.1 depicts an overview of the main components of UAFuzz. The inputs of the overall system are a set of initial seeds, the Program Under Test (PUT) (as a binary executable) and target locations extracted from the bug trace. The output is a set of unique bug-triggering inputs. The prototype is built upon AFL 2.52b [afl20a] and QEMU 2.10.0 for fuzzing, and the binary analysis platform Binsec [bin20] for lightweight static analysis. These two components share information such as target locations, time budget and fuzzing status.

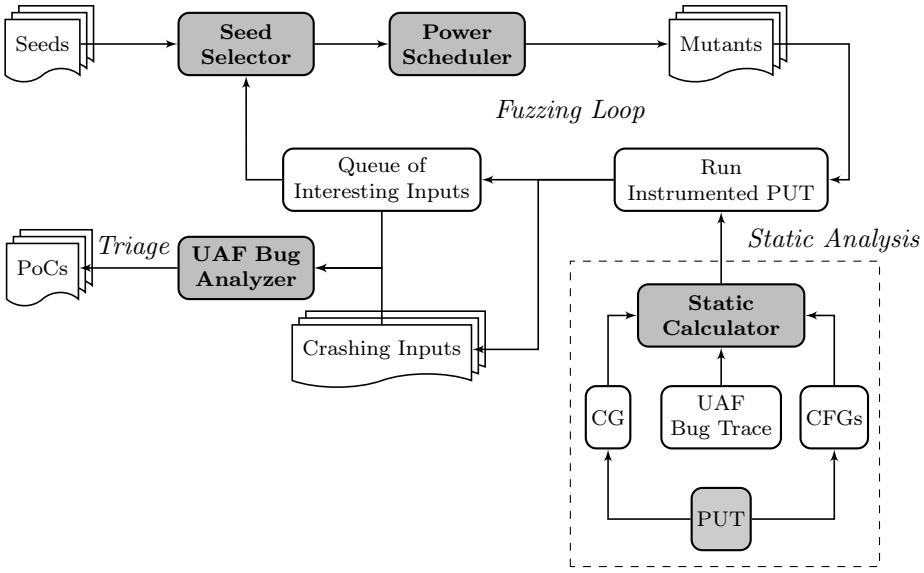


Figure 5.1: Overview of UAFuzz workflow.

- We have implemented a Binsec plugin computing *statically* distance and cut-edge information, consequently used in the instrumentation of UAFuzz – note that the call graph and the Control Flow Graph (CFG) are retrieved from the IDA Pro binary database (IDA Pro version 6.9 [ida20]). The static part takes about 2000 lines of Ocaml code;

- On the *dynamic* side, we have modified AFL-QEMU to track covered targets, dynamically compute seed scores and power functions, by adding another 3000 lines of C/C++ code;

- In the end, some scripts including 1000 lines of Python code and 1500 lines of Bash code automate the bug triaging and run the whole toolchain against the UAF fuzzing benchmark in Section 4.4, respectively.

UAFuzz

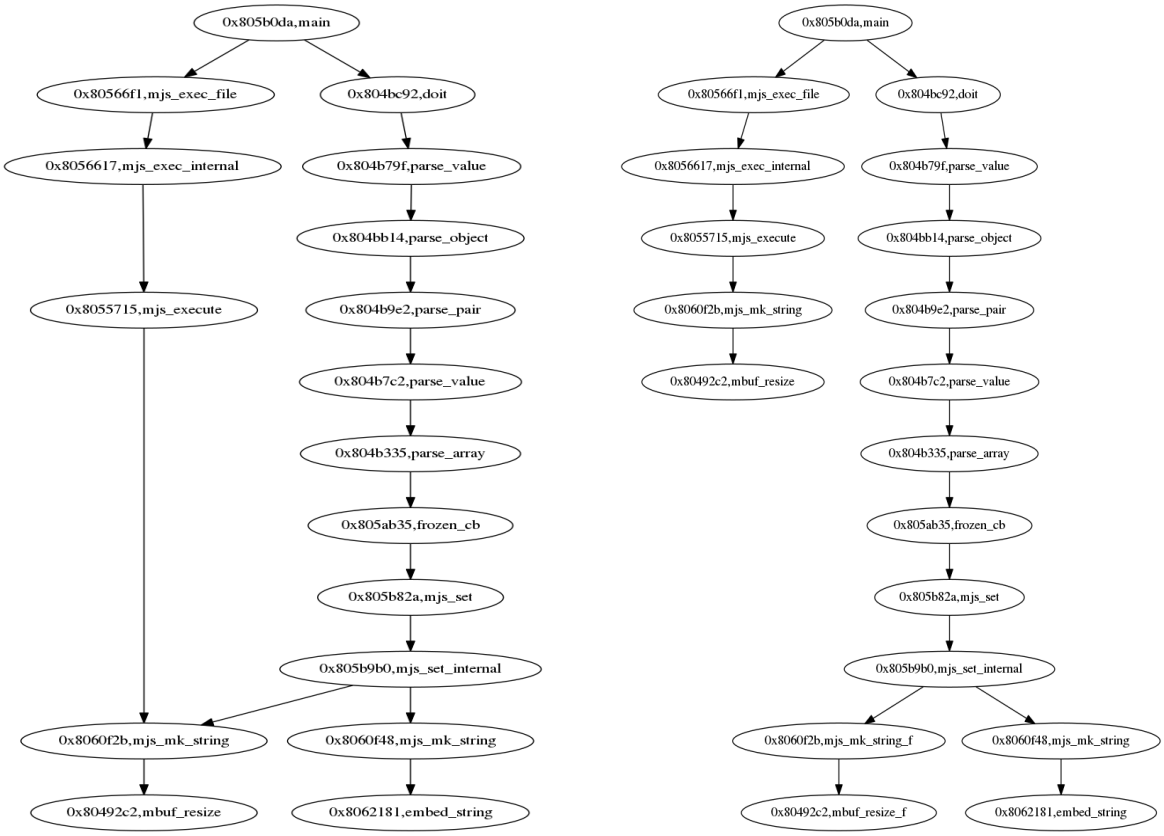
- **binsec/src**
  - **ida**: a plugin to import and process IDA's control-flow graphs and call graph
  - **uafuzz**: fuzzing code
    - **afl-2.52b**: core fuzzing built on top of AFL-QEMU
    - **uafuzz_*.ml(i)**: a plugin to compute static information and communicate with AFL-QEMU
- **scripts**: some scripts for building and bug triaging

Figure 5.2: Code structure of UAFuzz.

Availability We have made UAFuzz open source at https://github.com/strongcourage/uafuzz. The Github respository contains the source code of UAFuzz including Ocaml code of Binsec and C code of AFL, as well as the Python scripts that we used for building the fuzzer and triaging bugs in our experiments. Note that we use Ocaml-C bindings to call some important fuzzing functions, such as `afl-fuzz` and `afl-showmap`, implemented in AFL from Binsec. The code structure of UAFuzz is organized as in Figure 5.2.

5.2 Preprocessing

5.2.1 Bug trace generation



(a) The original calling tree. (b) The final calling tree.

Figure 5.3: Dynamic Calling Tree (DCT) of the program mjs generated by our preprocessing script.

The preprocessing script takes the tested x86 binary executable and the Valgrind's bug traces as inputs, then generates the UAF bug trace which is a sequence of target locations in the format `(basic_block_address, function_name)`. In addition, we also

output the DCT of the tested program, allowing users (e.g., developers) to have a better visualization of bug-triggering paths and buggy functions where the UAF events happen. However, there are some corner cases in which paths leading to buggy UAF events are not clearly identified, such as in `mjs-issue-73` in Figure 5.3. In particular, the two paths leading to *alloc* and *free* events share the similar nodes because of the common function `mjs_mk_string` is invoked multiple times by different functions, as shown in Figure 5.3a. We thus add redundant nodes with a suffix "_f" (noting that "_u" if nodes belong to the *use* path) in the final calling tree, as shown in Figure 5.3b to clearly distinguish the three paths and also support the process of bug trace generation by applying the preorder traversal algorithm on the DCT. Noting that adding redundant nodes in the dynamic calling tree and also in the bug trace has no impact on the calculation of metrics in the static analysis phase.

5.2.2 BinIda Plugin

The BinIda plugin is a part of Binsec version 0.3 [bin20]. The goal of this plugin is to extract information of the input binary in x86 using the disassembler IDA Pro, then construct the CFG that is represented by the data structure of Binsec. The Ida files contain the crucial information of the binary like functions, basic blocks and instructions in the following formats:

Function	start_addr; func_name
BasicBlock	start_addr; instructions; block_predecessors; block_successors; caller_call_addr – callee_start_addr – caller_return_addr
Instruction	addr; disasm; opcodes; block_start_addr; func_name

Figure 5.4: Formats of the files extracted using IDA Pro.

Concretely, we store the name and the entry point of each function. Then, for each basic block in a function, we collect the address of its first instruction, a list of its instructions, its predecessors, its successors and calling information. Finally, each instruction is associated with a basic block and a function to facilitate further processing. Furthermore, the calling information is more useful in constructing the interprocedural CFG for static analysis. As a basic block in CFGs produced by IDA Pro may contain many call instructions, as illustrated in Figure 5.5, we first need to split it into a sequence of blocks whose last instruction is a call or jump instruction. This processing step is indeed very important to make the static analysis relied on CFGs of IDA Pro consistent with the dynamic binary translation of QEMU, for example to keep track of covered edges or basic blocks during the fuzzing campaign. However, our tool chain shares the same problem with IDA Pro, that is the processing graphs are still incomplete due to indirect calls, thereby making our analysis less accurate.

Table 5.1 presents the detailed results of the plugin BinIda to preprocess the subjects in our evaluation benchmark in Table 4.3, discussed in Chapter 4. As our benchmark

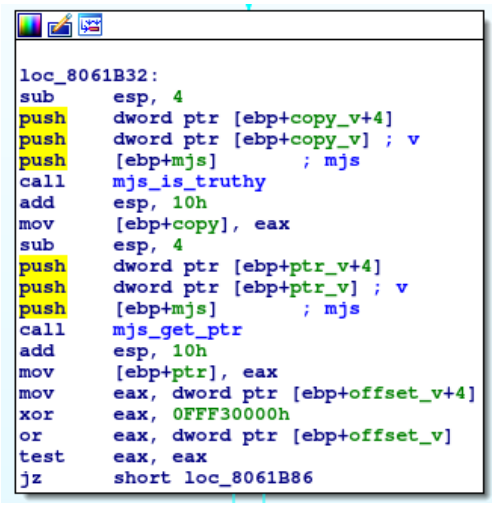


Figure 5.5: A basic block of `mjs_mkstr()` in the program mjs.

contains programs whose size vary from 26 Kb to 3.8 Mb, the size of the binary databases and the generated Ida files are relatively proportional to the size and the complexity of tested subjects. For example, for the most complex subject `cxxfilt` of Binutils (CVE-2016-4487), IDA Pro generates a database Idb file with size 24.1 Mb and BinIda outputs the biggest Ida file with size 29.5 Mb. Overall, the processing phase of BinIda is fast as BinIda takes less than 15 seconds for this step in the worst case scenario. Consequently, our tool chain is much faster than existing source-based directed fuzzers in the static analysis phase.

Table 5.1: Detailed results of BinIda in processing our evaluation benchmark in Table 4.3.

Bug ID	Program's Size	Database's size	Ida size	Processing Time (s)
giflib-bug-74	59 Kb	586 Kb	561 Kb	1.8
CVE-2018-11496	581 Kb	5.9 Mb	7.2 Mb	2.4
yasm-issue-91	1.4 Mb	12.3 Mb	12.4 Mb	6.7
CVE-2016-4487	3.8 Mb	24.1 Mb	29.5 Mb	14.2
CVE-2018-11416	62 Kb	523 Kb	293 Kb	1.5
mjs-issue-78	255 Kb	3.0 Mb	3.0 Mb	4.2
mjs-issue-73	254 Kb	3.0 Mb	2.9 Mb	4.5
CVE-2018-10685	576 Kb	5.9 Mb	7.3 Mb	2.7
CVE-2019-6455	604 Kb	6.3 Mb	6.8 Mb	4.6
CVE-2017-10686	1.8 Mb	11.7 Mb	7.4 Mb	7.5
gifsicle-issue-122	374 Kb	3.7 Mb	4.2 Mb	1.4
CVE-2016-3189	26 Kb	191 Kb	97 Kb	1.6
CVE-2018-20623	1.0 Mb	11.7 Mb	11.7 Mb	5.7

5.3 Core Fuzzing Engine

5.3.1 Debugging with afl-showmap

`afl-showmap` is a simple tool that runs the targeted binary and displays the contents of the trace bitmap in a human-readable form. In our toolchain, we also provide a way to facilitate the debugging process by invoking `afl-showmap`. Given a tested binary, a VALGRIND report and an input (e.g., an initial valid input or an input produced by the fuzzing), we can run the following command to obtain all input metric values of this input, as shown in Listing 5.1.

```
$ example.sh uafuzz 60 example.valgrind
...
UAFuzz afl-showmap 2.52b by <lcamtufgoogle.com>
[*] Executing '/home/dungnguyen/UAFuzz/tests/example/obj-uafuzz/example'...
- Program output begins -
[parse_distance] addr:  0x804853a, distance:  11.000000
[parse_distance] addr:  0x804852f, distance:  12.000000
[parse_distance] addr:  0x8048513, distance:  13.000000
[parse_distance] addr:  0x80484b5, distance:  13.000000
[parse_uaf_targets] addr:  0x8048513, fname:  main
[parse_uaf_targets] addr:  0x804849b, fname:  bad_func
[parse_uaf_targets] addr:  0x804854a, fname:  main
- Program output ends -
[+] total_distance:  37.000000, total_count:  41.000000, average_distance:  0.902439
[C] 0x804853f -> 0x804854a:  14845 -> hit 1 times
[C] 0x8048500 -> 0x8048513:  32633 -> hit 1 times
[+] nb_cut:  2, nb_uncut:  0
[+] trace_targets:  15, nb_reach:  4
[+] trace_uaf:  7, nb_uaf:  3
[+] Captured 40 tuples in 'out_file'.
[uafuzz:result] status:  0
```

Listing 5.1: Outputs of afl_showmap.

5.3.2 Overhead

Extended shared memory Since our fuzzer computes the seed metric values of each input produced at runtime, we extend the shared memory to store important current values, subsequently reduce the runtime overhead during fuzzing process. Overall, UAFUZZ uses 20 additional bytes of the shared memory as shown in Figure 5.6.

In order to make UAFUZZ aware of distance to targets, similar to state-of-the-art source-based directed fuzzer AFLGO, the shared memory that is passed by UAFUZZ during execution is extended by 16 bytes. Let D be the set of distance values corresponding to each basic block that is executed by the seed. The first eight additional bytes are used to accumulate the cumulative basic block distance values (i.e., $\sum_{d \in D} d$) as and when the seed is executed. These are followed by eight bytes that contain the count of accumulated

	8 bytes	8 bytes	4 bytes
Bit Map ...	Cumulative Distance	Number of Additions	Target Trace

Figure 5.6: UAFuzz shared memory – extended layout (x86-64)

distance values (i.e., $|D|$). Thus, those additional bytes allow us to compute the arithmetic mean of the distances of the exercised basic blocks as in Equation 3.3 (i.e., $\left(\sum_{d \in D} d\right) / |D|$).

The last extra four bytes (or 32 bits) represent the seed target trace for the current seed. Concretely, for the target similarity metric, as the maximum number of targets in a bug trace in our benchmarks is smaller than 32, each bit associates to one target and the bit is set if the current seed trace covers this target basic block. Thus, those four bytes allow us to quickly compute the target similarity metric of an input.

Furthermore, to compute the cut-edges coverage metric as in Equation 4.4, we can extract the hit counts of the exercised (non-) cut edges that are logged to the shared bitmap during execution, in which each byte represents an edge. To sum up, our seed metrics in UAFuzz were designed to be lightweight at runtime, allowing UAFuzz to have the same fuzzing speed (i.e., in executions per second) as the fuzzer baseline AFL-QEMU, as discussed in §4.4.4.

About performance of HawkeyeB in RQ1 HawkeyeB performs significantly worse than AFLGoB and UAFuzz in §4.4.3. We cannot compare HawkeyeB with Hawkeye as Hawkeye is not available. Still, we investigate that issue and found that this is mostly due to a large runtime overhead spent calculating the target similarity metric. Indeed, according to the Hawkeye original paper [CXL+18], this computation involves some *quadratic computation* over the *total number of functions* in the code under test. On our samples this number quickly becomes important (up to 772) while the number of targets (UAFuzz) remains small (up to 28). A few examples: `CVE-2017-10686`: 772 functions vs 10 targets; `gifsicle-issue-122`: 516 functions vs 11 targets; `mjs-issue-78`: 450 functions vs 19 targets. Hence, we can conclude that on our samples the performance of HawkeyeB are in line with what is expected from Hawkeye algorithm.

5.4 Examples

In the previous sections, we introduce the technical details of our directed fuzzer UAFuzz. In this section, we provide detailed instruction to run the whole toolchain in two practical applications: bug reproduction and patch testing.

5.4.1 Application 1: Bug Reproduction

We consider the simplified version of the motivating example discussed in Section 4.2 to illustrate the usage of UAFuzz in the bug reproduction. This example in Listing 5.2

contains a UAF bug due to a missing `exit()` call which could be triggered in a corner case if the first three bytes of the Proof-of-Concept (PoC) input are '`AFU`'. Concretely, the program reads a file and copies its contents into a buffer `buf`. A memory chunk pointed at by `p_alias` is allocated (line 20), then `p_alias` and `p` become aliased (line 21). The memory pointed by both pointers is freed in function `bad_func` (line 11). The UAF bug occurs when the freed memory is dereferenced again via `p` (line 26).

The corresponding Valgrind's output of the PoC is in Figure 5.9. Noting that a UAF bug could be triggered in a different way, for example with an input '`BFU`' by only exercising *then* branches of the last two conditional statements. However, in the bug reproduction setting, our final goal is to reproduce the UAF bug with the expected bug trace as in Figure 5.9. In other words, the fuzzer needs to generate an input exercising in sequence *then* branches of all conditional statements (line 19, 23 and 25). Given the stack traces, our fuzzer first generates the corresponding DCT as depicted in Figure 5.8. For instance, from the Valgrind's output, we know that there is a call of `malloc()` at address `0x804851C`, thus the root node of the DCT has the address `0x8048513` of a basic block containing this call instruction. As a result, we obtain the expected bug trace "`(0x8048513,main);(0x804853a,main);(0x804849b,bad_func);(0x804854a,main)`".

Figure 5.10 shows the call graph of the tested binary and CFGs of two important functions `main()` and `bad_func()` in the expected bug trace. From the CFG of `main()` and the bug trace, we can extract a list of (non-) cut edges of this example in the format `(type,block_address,successor_block_address)`, as shown in Figure 5.7. Then, during the fuzzing process, UAFuzz can easily identify how many (non-) cut edges are exercised by the current input and the hit counts of each edge from the bitmap, allowing the fuzzer to evaluate the reaching progress of this input at edge level with relatively low runtime overhead.

```
C,0x804853f,0x804854a
N,0x804853f,0x8048555
C,0x8048500,0x8048513
N,0x8048500,0x804852f
C,0x804852f,0x804853a
N,0x804852f,0x804853f
```

Figure 5.7: The identified (non-) cut edges of this example given the bug trace. C, N denotes cut and non-cut, respectively.

```c
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <fcntl.h>

int *p, *p_alias;
char buf[10];

void bad_func() {
  free(p); // exit() is missing
}

int main (int argc, char *argv[]) {
  int f = open(argv[1], O_RDONLY);
  read(f, buf, 10);
  p = malloc(sizeof(int));

  if (buf[0] == 'A') {
    p_alias = malloc(sizeof(int));
    p = p_alias;
  }
  if (buf[1] == 'F')
    bad_func();
  if (buf[2] == 'U')
    *p = 1;
  return 0;
}
```

Listing 5.2: A simple example.

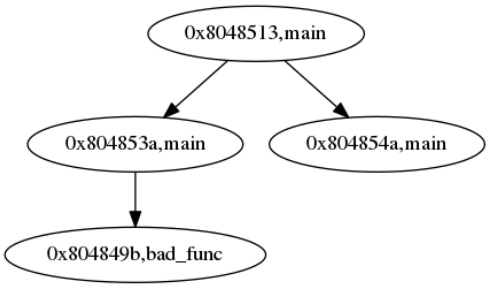
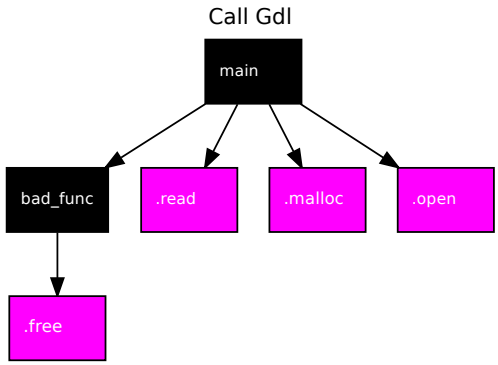


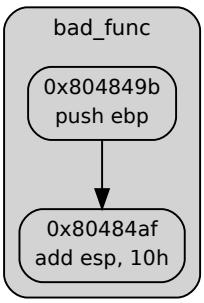
Figure 5.8: DCT of this example.

```
    // Stack trace for the bad Use
==27559== Invalid write of size 4
==27559== at 0x804854F: main (example.c:26)

    // Stack trace for the Free
==27559== Address 0x421d060 is 0 bytes inside a block of size 4 free'd
==27559== at 0x402D358:  free (in vgpreload_memcheck-x86-linux.so)
==27559== by 0x80484AE: bad_func (example.c:11)
==27559== by 0x804853E: main (example.c:24)

    // Stack trace for the Alloc
==27559== Block was alloc'd at
==27559== at 0x402C17C: malloc (in vgpreload_memcheck-x86-linux.so)
==27559== by 0x804851C: main (example.c:20)
```

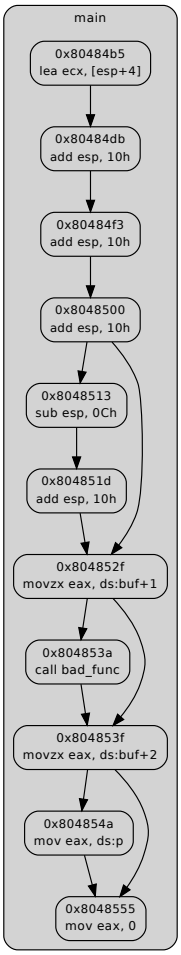
Figure 5.9: The stack traces of this example produced by VALGRIND.



(a) Call graph.



(b) CGF of bad_func().



(c) CFG of main().

Figure 5.10: Call graph and important CFGs (only show the first instruction of each basic block) of this example produced by the BinIda plugin.

Figure 5.11 illustrates the user interface of our fuzzer UAFuzz which is similar to AFL's. Although the maximum number of paths of the simple example is four (4), in this case, the total paths found by UAFuzz is five (5), which means there are 5 different inputs in the fuzzing queue. The reason, which is similar to what we explain in Section 4.2, is that UAFuzz determines that the fifth input (here '`AFU`' – the PoC) exercises in sequence the targets in the expected bug trace. Intuitively, although this kind of input does not increase the code coverage so far, it is definitely an interesting input that potentially triggers the desired bug. Thus, we mark all inputs exercising in sequence all target basic blocks in the

bug trace with "`,all`", and then run them under the profiling tool like VALGRIND in the triage phase to detect the PoC. It should be emphasized that both AFL-QEMU and even directed fuzzer AFLGO with targets at source-level can not detect this bug within 6 hours, while UAFUZZ can generate a PoC within minutes with the help of a VALGRIND's UAF report.

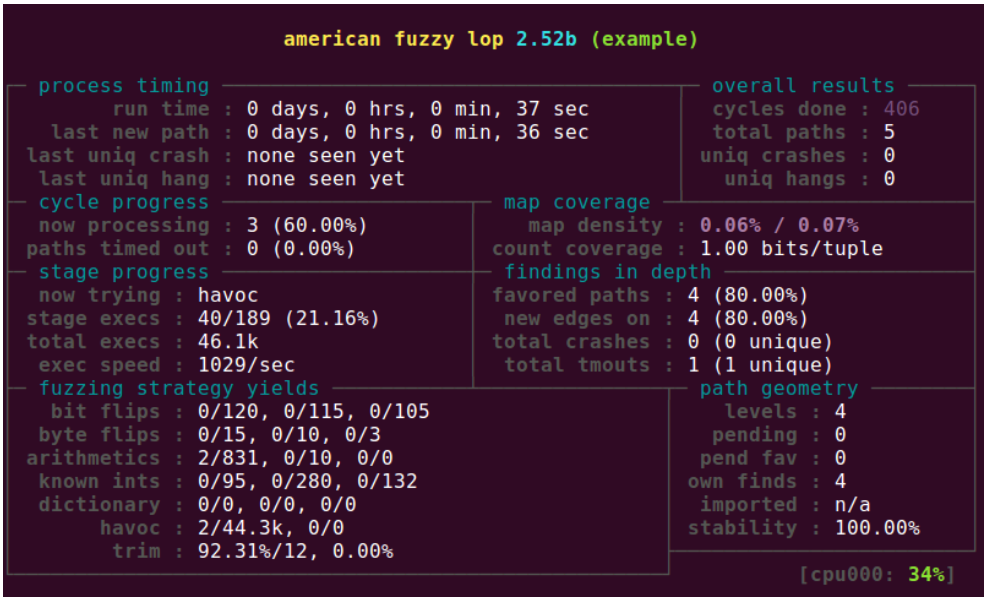


Figure 5.11: The user interface of UAFUZZ.

5.4.2 Application 2: Patch Testing

We use CVE-2018-6952 of GNU Patch to illustrate the importance of producing different unique bug-triggering inputs to favor the repair process. There was a double free in GNU Patch which has been fixed by developers (commit `9c98635`). However, by using the stack traces of CVE-2018-6952 in Figure 4.17, UAFUZZ discovered an incomplete bug fix CVE-2019-20633 of the latest version 2.7.6 (commit `76e7758`), with a slight difference of the bug trace.

Overall, the process is similar to the bug reproduction application, except that some manual work could be required in identifying the target UAF bug trace. More specifically, as the code has evolved (e.g., adding new features or fixing bugs), we may not find the corresponding basic block's addresses of CVE-2018-6952's VALGRIND output in the latest version of GNU Patch. What we can do is to automatically identify all call instructions of relevant buggy functions in the new/patched version as potential targets, and then select one of them to add to the bug trace. In this example, UAFUZZ is able to identify the correct UAF bug trace of our new bug CVE-2019-20633 from CVE-2018-6952's stack traces, as shown in Listing 5.3.

```
$ CVE-2019-20633.sh uafuzz 360 CVE-2018-6952.valgrind
```

```
[!] Cannot find BB address of (0x804c06c: main)
[!] Cannot find BB address of (0x805423c: another_hunk)
[!] Cannot find BB address of (0x805a821: savebuf)
[!] Cannot find BB address of (0x804c06c: main)
[!] Cannot find BB address of (0x804c06c: main)
[+] Alloc path: [['0x804c38d', 'main'], ['0x80555bd', 'another_hunk'], ['
   0x805ae62', 'savebuf']]
[+] Free path: [['0x804c38d', 'main'], ['0x80531f2', 'another_hunk']]
[+] Use path: [['0x804c38d', 'main'], ['0x80531f2', 'another_hunk']]
[+] Possible targets:
{('0x804c06c', 'main'): [('0x804c38d', 'main')],
('0x8052e11', 'another_hunk'): [('0x80546f3', 'another_hunk'),
('0x80557e2', 'another_hunk'),
('0x805560a', 'another_hunk'),
('0x8055948', 'another_hunk'),
('0x80559fc', 'another_hunk'),
('0x80559e9', 'another_hunk'),
('0x80557f5', 'another_hunk'),
('0x80556d9', 'another_hunk'),
('0x8053150', 'another_hunk'),
('0x80531f2', 'another_hunk')],
('0x805423c', 'another_hunk'): [('0x805619e', 'another_hunk'),
('0x8055597', 'another_hunk'),
('0x80558a8', 'another_hunk'),
('0x80564e7', 'another_hunk'),
('0x8054564', 'another_hunk'),
('0x80542dc', 'another_hunk'),
('0x8054167', 'another_hunk'),
('0x80555bd', 'another_hunk')],
('0x805a821', 'savebuf'): [('0x805ae62', 'savebuf')]}
[+] UAF bug trace: ['0x804c38d,main', '0x80555bd,another_hunk', '0
   x805ae62,savebuf', '0x80531f2,another_hunk']
...
```

Listing 5.3: UAFuzz's output when fuzzing the latest version of GNU Patch with the timeout 6 hours.

5.5 Conclusion

In this chapter, we have introduced our directed fuzzer UAFuzz at https://github.com/strongcourage/uafuzz in details, especially its principal components like the preprocessing component, the core fuzzing engine and also our UAF fuzzing benchmark at https://github.com/strongcourage/uafbench. Furthermore, we have explained step by step how to use UAFuzz to detect UAF vulnerabilities in two security applications: bug reproduction and patch testing.

In the future, we can improve UAFuzz in several directions. First, in the preprocessing phase, we can use other open-source disassemblers like Radare2 [r220] to generate important graphs, like the call graph and the CFGs, of the tested binary. Second, AFLplus-

plus [FMEH20, afl20h] was created initially to incorporate all the best features developed in the years for the fuzzers in the AFL family, thus, if UAFuzz were built on top of AFLplusplus, it could boost the fuzzing performance of UAFuzz in general. Finally, combining UAFuzz with the binary-level static analyzer Graphs of Use-After-Free Extracted from Binary (GUEB) [gue20] in a hybrid manner could detect more UAF vulnerabilities.

Chapter 6

Typestate-guided Directed Fuzzing

Contents

6.1 Introduction **79**
6.2 The TypeFuzz Approach **81**
6.2.1 Different Bug Characteristics 81
6.2.2 Adapted Techniques 82
6.3 Evaluation **83**
6.3.1 Research Questions 83
6.3.2 Evaluation Setup 84
6.3.3 Bug-reproducing Ability (RQ1) 85
6.3.4 Crash Triage (RQ2) 86
6.3.5 Target Reaching (RQ3) 87
6.4 Patch Testing **89**
6.5 Conclusion **89**

In this chapter, we introduce TypeFuzz, a binary-level directed fuzzer built on top of UAFuzz specializing to detect common typestate vulnerabilities, such as buffer overflows (CWE-121, CWE-122) and NULL pointer dereference (CWE-476). We then evaluate the effectiveness and efficiency of TypeFuzz on the benchmarks used in the state-of-the-art Directed Greybox Fuzzing (DGF) work and real-world programs as well.

6.1 Introduction

Classic memory corruptions identified by Common Weakness Enumeration (CWE) like buffer overflows (CWE-121, CWE-122) [CPM+98, HSNB13], NULL pointer dereference (CWE-476) [HP07,FMRS12] or integer overflows (CWE-190) [WWLZ09,MLW09,DLRA15] have been well studied. In contrast, recent vulnerability classes such as UAF (CWE-415, CWE-416) or type confusion have not received much attention in the literature. In the previous chapters, we have introduced our directed fuzzer UAFuzz tailored to complex UAF bugs, by carefully tuning several of its key components to the bug-triggering conditions of

UAF that rely on the sequence of finite-state machine $\langle alloc \rightarrow free \rightarrow use \rangle$. Since other types of bugs can also be triggered by the violation of typestate properties [SY86], we aim to investigate the generality of our proposed directed techniques in Chapter 4 against more popular memory corruption bugs.

Typestate properties can aid program understanding, define type systems [DF04] that prevent programmers from causing typestate errors or even derive static analysis [FGRY03, FYD+08] to verify whether a given program violates typestate properties, especially in formal verification. For example, the sequence of finite-state machine $\langle nullify \rightarrow dereference \rangle$ is a witness of triggering the NULL pointer dereference bug. However, typestate verification problem becomes NP-hard for complex programs, for example with maximum aliasing width of three and aliasing depth of two, as shown by Field et al [FGRY03], preventing it to be practically applicable on large programs. Recent work proposed new approaches to applying typestate analysis by incorporating it into software testing techniques. Hua et al. proposed Machine Learning (ML)-guided typestate analysis for static UAF detection by leveraging ML techniques to tackle the problem of high overhead of typestate analysis, making it scalable to real-world programs [YSCX17]. Recently, UAFL [WXL+20] – another independent research effort specialized coverage-guided fuzzing to detect UAFs in source code by finding operation sequences potentially violating a typestate property and then guiding the fuzzing process to trigger property violations.

Overview In general, TypeFuzz is built on top of UAFuzz in the hope of detecting typestate bugs. Similar to UAFuzz, TypeFuzz is made out of several components including seed selection, power schedule, and crash triage, as illustrated in Figure 6.1.

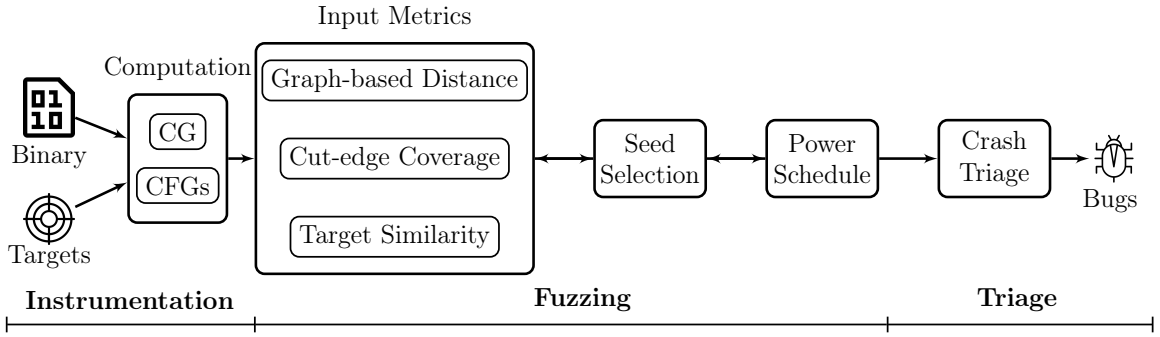


Figure 6.1: Overview of TypeFuzz.

Our intuition behind TypeFuzz is to leverage the *relationship among target locations* in the expected bug trace to accelerate detecting complex behavioral bugs. Given the expected bug trace, we still combine three dynamic *ordering-awareness seed metrics* to evaluate an input produced by the fuzzer at runtime at different granularity levels, e.g., function, edge and basic block. Our seed selection strategy then favors seeds covering more targets at runtime and their energy is determined via our power schedule. Finally, we take

advantage of our previous metrics to pre-identify likely-Proof-of-Concept (PoC) inputs that are sent to a profiling tool (e.g., VALGRIND [NS07]) for detecting the real PoC.

Contributions We summarize the contributions as follows.

- We study bug-triggering conditions of different typestate bugs and tailor the directed fuzzing strategies of UAFUZZ into TYPEFUZZ to detect popular memory-related bugs in the C/C++ programs, such as buffer overflows.

- We evaluate TYPEFUZZ with real-world programs in two practical settings, demonstrating that TYPEFUZZ outperforms state-of-the-art competitors in reproducing known bugs and in finding new bugs (7 CVEs were assigned and all bugs were fixed). Furthermore, our evaluations show that TYPEFUZZ is also effective in reaching a target basic block, especially in cases where the complete bug trace is given.

6.2 The TypeFuzz Approach

6.2.1 Different Bug Characteristics

Runtime behavior Different bugs have different runtime behaviors. While UAF bugs usually fail silently without segmentation fault, buffer overflows or NULL pointer dereference crash programs frequently. Thus, this characteristic affects the triage phase in the fuzzing workflow.

Bug stack trace Different bugs have different stack traces produced by the profiling tools, such as AddressSanitizer or VALGRIND, given the bug-triggering input. Considering the CVE-2018-4488 which is a NULL pointer dereference of the Binutils program `cxxfilt`, Figure 6.2 illustrates the stack trace produced by VALGRIND. Our bug trace that is extracted from this stack trace, contains a sequence of target locations in the same format "`function,address_of_block`" used in UAFUZZ, as shown in Listing 6.1.

```
==32611== Invalid write of size 4
==32611== at 0x813A8E5:  register_Btype (cplus-dem.c:4319)
==32611== by 0x8137611:  demangle_class (cplus-dem.c:2594)
==32611== by 0x81355D8:  demangle_signature (cplus-dem.c:1490)
==32611== by 0x8134D07:  internal_cplus_demangle (cplus-dem.c:1203)
==32611== by 0x8134466:  cplus_demangle (cplus-dem.c:886)
==32611== by 0x8049A23:  demangle_it (cxxfilt.c:62)
==32611== by 0x8049E21:  main (cxxfilt.c:227)
==32611== Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Figure 6.2: The stack trace of CVE-2016-4488 produced by VALGRIND.

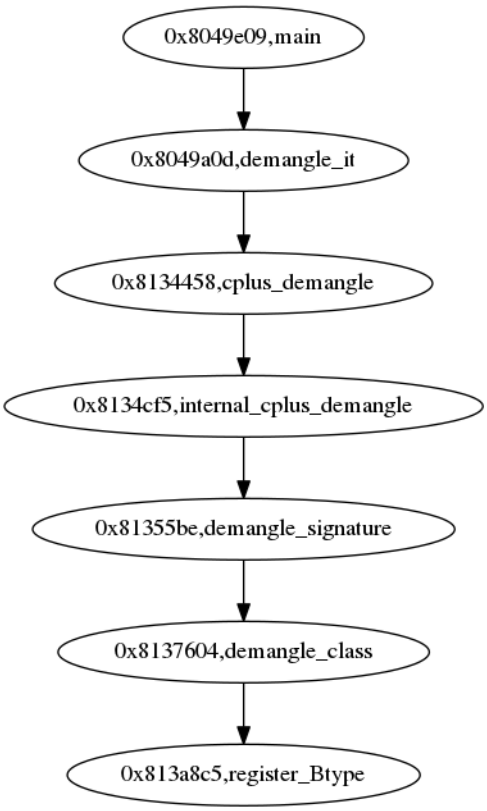


Figure 6.3: DCT of CVE-2016-4488.

```
(0x8049e09,main);(0x8049a0d,demangle_it);(0x8134458,cplus_demangle);
(0x8134cf5,internal_cplus_demangle);(0x81355be,demangle_signature);
(0x8137604,demangle_class);(0x813a8c5,register_Btype)
```

Listing 6.1: The expected bug trace of CVE-2016-4488.

6.2.2 Adapted Techniques

TypeFuzz takes the tested binary, the expected bug trace and a set of initial test cases as inputs and produces PoCs that trigger the desired bug. However, as discussed in detail above, different types of bugs have different characteristics that have impact on our designs of the key fuzzing components in a more general context. We thus discuss our adaption of directed fuzzing techniques proposed in Chapter 4 in TypeFuzz. Overall, we need to slightly modify all three principal phases of DGF to tackle the problem of finding typestate vulnerabilities.

Instrumentation In the first step, we still employ the plugin BinIda to generate the important graphs of the tested binary. However, in TypeFuzz, we do not need the bug trace flattening step (in §4.3.1) because there is only one stack trace produced by Val-

GRIND, as shown in Figure 6.2. Therefore, we consider this unique stack trace as the bug trace, and also generate Dynamic Calling Tree (DCT) which represents a path starting from `main()` and leading to the crashing point (a.k.a, the last location in the bug trace). Finally, we perform some static analysis to precompute important information for fuzzing, such as distance values or edges labeling.

Graph-based distance metric In TYPEFUZZ, we prioritize call trace leading to the crashing function. In this case, we favor call edges between two functions belonging to paths that can reach the crashing function in the call graph.

$$\Theta(f_a, f_b) \triangleq \begin{cases} \beta & \text{if } f_a \rightarrow f_b \text{ can reach the crashing function} \\ 1 & \text{otherwise} \end{cases} \tag{6.1}$$

In our experiments, we use the following $\Theta(f_a, f_b)$ function, with a value of $\beta = 0.25$, like in UAFUZZ. Finally, we define our edge weight:

$$w_{UAFuzz}(f_a, f_b) \triangleq w_{Hawkeye}(f_a, f_b).\Theta(f_a, f_b) \tag{6.2}$$

Cut-edge coverage metric This metric is directly applied in TYPEFUZZ without any modification, as discussed in §4.3.4.1.

Target similarity metric In UAFUZZ, we have two interesting traces: the sequence of UAF events and the bug trace itself. In contrast, there is only the bug trace in TYPEFUZZ. Therefore, in this case, our target similarity metric leverages the combination of Prefix (**P**) and Bag (**B**) values of the current input execution trace towards the expected bug trace. Hence, the *P-B* metric is defined as:

$$t_{P-B}(s, T) \triangleq \langle t_P(s, T), t_B(s, T) \rangle \tag{6.3}$$

Triage As the typestate bugs like buffer overflow and NULL pointer dereference usually crash the tested program, we are therefore interested in the crashing inputs. In other words, we only triage the crashing inputs in the `/crashes` directory. Furthermore, our target similarity metric allows us to identify inputs in the fuzzing queue that trigger in sequence all target locations in the expected bug trace. In case where the bugs fail silently, TYPEFUZZ still takes advantage of this seed metric to pre-identify likely-PoC inputs and then only triages such kinds of potential inputs, like in UAFUZZ.

6.3 Evaluation

6.3.1 Research Questions

In the bug reproduction setting, to evaluate the effectiveness and efficiency of our approach, we investigate the following research questions:

RQ1. Bug-reproducing Ability Can TypeFuzz outperform other directed fuzzing techniques in terms of typestate bug reproduction in executables?

RQ2. Crash Triage Can TypeFuzz find more correct crashing inputs than other fuzzers?

RQ3. Target Reaching Is TypeFuzz effective at reaching a specific target location in the bug trace?

It is noted that we skip two research questions **RQ2 - Overhead** and **RQ4 - Individual Contribution** in §4.4.1 as the results are straight-forward. First, similar to UAFuzz, the overhead of TypeFuzz is relatively small as both fuzzers have the same preprocessing component. Second, TypeFuzz uses the best configurations of UAFuzz. Furthermore, we add a new research question to evaluate the *target reaching ability* of existing fuzzers, as in Hawkeye [CXL+18] and ParmeSan [ÖRBG20].

6.3.2 Evaluation Setup

Evaluation fuzzers Similar to the experiments in Chapter 4, we mainly compare TypeFuzz with state-of-the-art directed fuzzers AFLGoB and HawkeyeB and also with coverage-guided fuzzer AFL-QEMU.

Benchmarks Table 6.1 shows the benchmarks we use in our evaluations for crash reproduction. As typestate bugs like buffer overflows can be easily found in comparison with UAF, our main goal is to evaluate TypeFuzz with diverse real-world programs used in existing (directed) fuzzing work and various types of bugs to make our evaluations more thoughtful.

- We reuse the benchmarks for crash reproduction that were used in existing directed fuzzing work [BPNR17, CXL+18]. Concretely, we first use the GNU Binutils benchmark[1] in AFLGo' paper [BPNR17]. Second, we also use several bugs of the restricted JavaScript engine `mjs` in Hawkeye's paper [CXL+18], which contains a single source file in order to avoid some issues in the instrumentation phase of AFLGo. It is worth noting that we skip some UAF bugs in these benchmarks that have been used for evaluations of UAFuzz in Chapter 4;
- As discussed in §4.4.7, UAFuzz reported some typestate bugs when fuzzing programs to find new UAF bugs in the patch-oriented testing. We thus select some of those bugs in order to evaluate TypeFuzz;
- Finally, we also collect recent typestate bugs reported by existing coverage-guided greybox fuzzers, such as Profuzzer [YWM+19], to increase the diversity of our tested programs (e.g., evix2, openjpeg, libming).

[1]Here we skipped old CVEs in libpng and we failed to reproduce CVE-2016-4491 due to lack of bug trace in the bug report.

Table 6.1: Overview of our evaluation benchmark.

Bug ID	Program		Bug	
	Project	Size	Type	Crash
CVE-2016-4488	Binutils	3.8 Mb	Invalid write	✓
CVE-2016-4489	Binutils	3.8 Mb	Integer overflow	✓
CVE-2016-4492	Binutils	3.9 Mb	Stack overflow	✓
CVE-2016-4493	Binutils	3.9 Mb	Invalid read	✓
mjs-issue-57	mjs	255 Kb	Integer overflow	✓
mjs-issue-69	mjs	254 Kb	Integer overflow	✓
mjs-issue-77	mjs	254 Kb	Heap buffer overflow	✓
CVE-2019-20629	GPAC	545 Kb	Heap buffer overflow	✓
CVE-2019-20630	GPAC	545 Kb	Heap buffer overflow	✓
fontforge-bug-4267	FontForge	578Kb	NULL pointer dereference	✓
boolector-bug-90	Boolector	79 Kb	NULL pointer dereference	✓
CVE-2017-17723	exiv2	4.2 Mb	Heap buffer overflow	✓
CVE-2018-5785	openjpeg	2.1 Mb	Heap buffer overflow	✓
CVE-2018-5294	libming	1.7 Mb	Integer overflow	✓

Evaluation configurations Similar to the configurations used in Chapter 4, we follow the recommendations for fuzzing evaluations [KRC+18] and use the same fuzzing configurations and hardware resources for all experiments. Experiments are conducted 10 times with a time budget depending on the PUT. We use as input seed either an empty file or existing valid files provided by developers. We do not use any token dictionary. All experiments were carried out on an Intel Xeon CPU E3-1505M v6 @3.00GHz CPU with 32GB RAM and Ubuntu 16.04 64-bit.

6.3.3 Bug-reproducing Ability (RQ1)

Protocol We compare the different fuzzers on the popular benchmarks used in existing work [BPNR17, CXL+18, ÖRBG20] using *Time-to-Exposure* (TTE), i.e. the time elapsed until first bug-triggering input, and *number of success runs* in which a fuzzer triggers the bug, as in §4.4.3. In case a fuzzer cannot detect the bug within the time budget, the run's TTE is set to the time budget. Following existing work [BPNR17, CXL+18], we also use the *Vargha-Delaney statistic* ($\hat{A}_{12}$) metric [VD00] to assess the confidence that one tool outperforms another.

Results Figure 6.4 presents a consolidated view of the results including total success runs and TTE – we denote by μTTE the average TTE observed for each sample over 10 runs. Table 6.2 summarizes the fuzzing performance of 4 binary-based fuzzers against the evaluated benchmark by providing the total number of success runs and the max/min/average/median values of $\hat{A}_{12}$.

Figure 6.4 and Table 6.2 show that UAFuzz outperforms the other fuzzers both in total success runs (vs. 2nd best HawkeyeB: +42% in total) and in TTE (vs. 2nd best

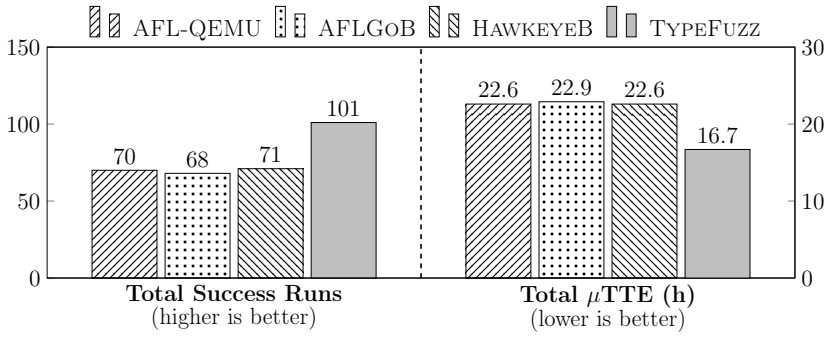


Figure 6.4: Summary of fuzzing performance (RQ1).

HawkeyeB: 1.4× in total). The $\hat{A}_{12}$ value of UAFuzz against other fuzzers is close to the conventional large effect size 0.71 [VD00], as shown in Table 6.2, especially vs. AFL-QEMU with median: 0.72 and max: 1.00.

Table 6.2: Summary of bug reproduction of TypeFuzz compared to other fuzzers against our fuzzing benchmark. Statistically significant results $\hat{A}_{12} \geq 0.71$ are marked as bold.

Fuzzer	Success Runs	$\hat{A}_{12}$			
		Mdn	Avg	Min	Max
AFL-QEMU	70 (+44%)	**0.72**	0.65	0.13	**1.00**
AFLGoB	68 (+46%)	0.57	0.65	0.35	**1.00**
HawkeyeB	71 (+42%)	0.62	0.66	0.25	**1.00**
TypeFuzz	101	–	–	–	–

Answer to RQ1: In bug reproduction, TypeFuzz outperforms state-of-the-art directed fuzzers in terms of total success runs and time to bug exposure.

6.3.4 Crash Triage (RQ2)

Protocol As all evaluated bugs in our benchmark (in Table 6.1) cause crashes, we consider only the total number of *crashing inputs* (not the triaging inputs sent to the triaging step as discussed in §4.4.5). Note that in the triage phase of bug reproduction, we need to verify whether a fuzzer triggers the expected bug with the expected bug trace, because a crashing input may trigger a different bug that we are not interested in to reproduce. Here, we simply run the buggy program with all crashing inputs under the profiling tool Valgrind, then compare the Valgrind's outputs with the expected bug traces to identify correct PoCs.

Results The detailed results are presented in Table 6.3. Overall, TypeFuzz found more correct crashing inputs than other fuzzers (e.g., vs. 2nd best AFLGoB: +22% in total).

Table 6.3: Average number of correct crashing inputs of 4 fuzzers against our tested subjects. Numbers in red are the best values.

Bug ID	AFL-QEMU	AFLGoB	HawkeyeB	TypeFuzz
CVE-2016-4488	1.3	0.5	0.8	2.3
CVE-2016-4489	1.1	1.6	1.5	1.5
CVE-2016-4492	0.3	0.4	0.2	0.9
CVE-2016-4493	0.3	0.4	0.2	0.9
mjs-issue-57	2.4	2.7	2.5	2.8
mjs-issue-69	0.2	0	0.2	0.8
mjs-issue-77	0.5	0	0.3	1.1
CVE-2019-20629	2.6	3.1	2.5	3.1
CVE-2019-20630	1.3	1.1	0.9	1.9
fontforge-bug-4267	3.1	5.3	5.1	5
boolector-bug-90	3.1	3.5	3.9	3.6
CVE-2017-17723	0	1.1	0	1.4
CVE-2018-5785	1.2	1.4	1	1.1
CVE-2018-5294	1.4	1.6	0.9	1.3
Total	**18.7**	**22.8**	**20.2**	**27.8**

Furthermore, as the number of crashing inputs produced by all 4 fuzzers is very small, those fuzzers spent relatively the same time in the triaging phase (in seconds in total). To summarize, the number of correct crashing inputs is also proportional to the fuzzing performance of 4 fuzzers, such as the number of success runs, as discussed in **RQ1**.

Answer to RQ2: TypeFuzz finds more correct crashing inputs, that produce the correct bug trace, in comparison with other fuzzers.

6.3.5 Target Reaching (RQ3)

In order to trigger the desired bug, the fuzzers first need to reach the buggy location as fast and often as possible. In other words, reaching quickly specific "*hard-to-reach*" locations implies the effectiveness of driving the fuzzer at runtime. Therefore, this metric is also an important criterion for measuring directed fuzzers' capabilities. Like existing work [CXL+18, ÖRBG20], we choose the popular benchmark Google Fuzzer TestSuite [gft20] that contains various types of bugs of real-world projects. Although this benchmark is widely used to assess fuzzing effectiveness of coverage-guided fuzzers on code coverage in the fuzzing literature, it also contains some bugs to test fuzzers' abilities in term of covering a target locations. Here we manually target a number of known hard-to-reach locations in those bug-free programs to indicate that the relevant targets have been reached.

Table 6.4 and Table 6.5 show the average *Time-to-Reach* (TTR) of 4 fuzzers against our tested subjects, given only one target basic block and a full bug trace, respectively. In Table 6.5, we also add the difference values of TTR in two settings of 3 directed fuzzers in parentheses, as the TTRs of the coverage-guided fuzzer AFL-QEMU remain unchanged regardless of the number of given targets. Note that while the *Time-to-Exposure* (TTE) is relevant to code coverage as complex bugs can only be triggered with special path

Table 6.4: Average TTR of 4 fuzzers against our tested subjects, given *only one target basic block*. Numbers in red are the best μTTRs.

Bug ID	AFL-QEMU	AFLGoB	HawkeyeB	TypeFuzz
CVE-2016-5180	52	25	19	15
CVE-2015-8317	10723	8881	9400	6538
CVE-2014-0160	5673	9632	8274	5890
CVE-2015-3193	245	90	143	145
freetype2-2017	30	25	13	6
libpng-1.2.56-#1	251	265	102	78
libpng-1.2.56-#2	452	267	198	231
libjpeg-turbo-07-2017	9673	6734	7352	10352
lcms-2017-03-21	561	722	1362	1021
libarchive-2017-01-04	4528	7139	7012	5613
Total μTTR (h)	**8.9**	**9.4**	**9.4**	**8.3**

Table 6.5: Average TTR in seconds of 4 fuzzers against our tested subjects, given *a full bug trace*. Numbers in red are the best μTTRs. The difference values of 3 directed fuzzers compared to Table 6.4 are in parentheses.

Bug ID	AFL-QEMU	AFLGoB	HawkeyeB	TypeFuzz
CVE-2016-5180	52	15 (-40%)	14 (-26.3%)	7 (-53.3%)
CVE-2015-8317	10723	8980 (+1.1%)	7443 (-20.8%)	4283 (-34.5%)
CVE-2014-0160	5673	11352 (+17.8%)	7734 (-6.5%)	3734 (-36.6%)
CVE-2015-3193	245	135 (-33.3%)	178 (+19.7%)	188 (+29.6%)
freetype2-2017	30	24 (-4%)	8 (-38.5%)	8 (+33.3%)
libpng-1.2.56-#1	251	195 (-26.4%)	142 (+39.2%)	56 (-28.2%)
libpng-1.2.56-#2	452	263 (-1.5%)	238 (+20.2%)	145 (-37.2%)
libjpeg-turbo-07-2017	9673	5473 (-18.7%)	7342 (-0.1%)	4550 (-56%)
lcms-2017-03-21	561	922 (+27.7%)	1227 (-9.9%)	1025 (+0.4%)
libarchive-2017-01-04	4528	4593 (-35.7%)	7182 (+2.42%)	3612 (-35.6%)
Total μTTR (h)	**8.9**	**8.9**	**8.8**	**4.9**

conditions, TTR is actually how long a fuzzer spends covering the specific target location at the first time.

Results In Table 6.4, TypeFuzz's improvements against other fuzzers in covering a specific target location are not obvious, and for several cases, it performs worse. In Table 6.5, we can clearly observe that the acceleration on target reaching ability is significant, as TypeFuzz outperforms other fuzzers in 8 out of 10 cases with a speed up of 1.8×. One notable result is `libjpeg-turbo-07-2017`, as given the bug trace, TypeFuzz saves roughly 56% of the TTR to become the best fuzzer reaching the target in this setting. Those results show that our dynamic fuzzing strategies are effective in detecting bugs, especially in cases where we have a complete bug trace. From the results, we can conclude that our ordering-awareness seed metrics that consider the relationship among target locations are effective to guide the fuzzer at runtime.

Answer to RQ3: Given a full bug trace, TypeFuzz performs better than other fuzzers in 8/10 cases in reaching a target location, and achieve significantly a speedup of 1.8× compared to other fuzzers.

6.4 Patch Testing

Similar to patch testing in UAFuzz in §4.4.7, we leverage bug stack traces of *known* bugs to guide testing on the more recent version of the Program Under Test (PUT), in the hope of finding buggy patches *and* performing stress testing on *a priori* fragile parts of the code. Here, we focus mainly on 2 widely-used open-source C/C++ programs that have been well fuzzed by Google OSS-Fuzz [oss20a] and other fuzzing projects. While Binutils is a collection of binary analysis tools and has almost one million lines of code, OpenEXR provides the specification and reference implementation of the EXR file format, the professional-grade image storage format of the motion picture industry. Both are well-maintained by the developers.

Results Overall UAFuzz has found and reported **7 new bugs**, including 2 buffer overflows, 4 NULL pointer dereferences and an invalid read, in critical libraries Binutils and OpenEXR (details in Table 6.6). All 7 bugs have been fixed by the vendors and 7 CVEs were assigned as they can cause a denial of service of programs which use those libraries.

Table 6.6: Summary of zero-day vulnerabilities reported by our fuzzer TypeFuzz. HBO, NPD denote heap buffer overflow and NULL pointer dereference, respectively.

Program	Code Size	Version (Commit)	Bug ID	Vulnerability Type	Crash	Vulnerable Function	Status	CVE
readelf	1.0 M	2.34 (f717994)	#25822	Invalid read	✓	process_symbol_table	Fixed	CVE-2020-16591
addr2line	4.0 M	2.34 (95a5156)	#25827	NPD	✓	scan_unit_for_symbols	Fixed	CVE-2020-16593
objdump	5.3 M	2.34 (8e4979a)	#25840	NPD	✓	debug_get_real_type	Fixed	CVE-2020-16598
nm-new	6.7 M	2.34 (1619720)	#25842	NPD	✓	_bfd_elf_get_symbol_version_string	Fixed	CVE-2020-16599
OpenEXR	187 K	2.3.0 (9410823)	#491	HBO	✓	chunkOffsetReconstruction	Fixed	CVE-2020-16587
			#493	NPD	✓	generatePreview	Fixed	CVE-2020-16588
			#494	HBO	✓	writeTileData	Fixed	CVE-2020-16589

TypeFuzz has been proven effective in leveraging existing bug traces to find 7 new bugs in error-prone software libraries that are patched more often than not. All 7 bugs were quickly fixed by the developers and were assigned CVEs.

6.5 Conclusion

In this chapter, we first have introduced TypeFuzz, which is built on top of UAFuzz by adapting directed fuzzing techniques proposed in Chapters 4 and 5 in a general context to detect common typestate vulnerabilities in binary code. Then, we have evaluated its effects against several state-of-the-art (directed) greybox fuzzers on some popular benchmarks of

real-world programs. To summarize, our evaluation has shown that our ordering-awareness seed metrics are effective not only in guiding the fuzzer to reproduce known bugs given a bug trace or find new vulnerabilities in critical libraries, but also in reaching a specific target location.

Chapter 7

Conclusion

Contents

7.1 Summary **91**
7.1.1 Research problems 91
7.1.2 Scientific contributions 92
7.1.3 Technical contributions 93
7.2 Perspectives **93**

In this chapter, we summarize the research problems, our proposed techniques, our achieved results and the limitations of the techniques proposed in this thesis. We also discuss some interesting follow-up directions in future work.

7.1 Summary

7.1.1 Research problems

Fuzzing, especially Coverage-guided Greybox Fuzzing (CGF), is a popular security testing technique consisting in generating massive amounts of random inputs, very effective in triggering bugs in real-world programs. On the other hand, Directed Greybox Fuzzing (DGF) aims to perform stress testing on pre-selected potentially vulnerable target locations, therefore it has many practical applications to different security contexts, such as bug reproduction and patch testing. Despite tremendous recent progress to tackle various fuzzing challenges [BCR21, MHH+19] in the past few years, finding complex vulnerabilities, such as Use-After-Free (UAF), is still hard for existing (directed or not) greybox fuzzers as bug-triggering paths may satisfy very *specific properties* of specific bug classes. In order to detect specific bugs more efficiently, we first need to perform further analysis to acquire a better understanding on how to trigger the target bugs, and then propose desired solutions to satisfy complex bug-triggering conditions. Furthermore, finding bugs in binary code is also needed since the source code of some critical programs is unavailable or relies on third-party libraries.

In summary, this thesis aims to *develop effective directed fuzzing techniques to detect complex typestate vulnerabilities, like* UAF*, in binary code of real-world programs in diverse security applications.*

7.1.2 Scientific contributions

A survey on directed fuzzing Our first contribution is to provide a systematic overview of the state-of-the-art DGF including its applications, its differences compared to coverage-guided fuzzing, a formal definition of the problem, an overview of existing solutions and current limitations.

Directed fuzzing for complex vulnerabilities The second principle contribution of this thesis is the design, implementation and testing of UAFuzz, which is the *first directed* greybox fuzzing framework tailored to detecting UAF vulnerabilities in binary given only the bug stack trace. We have shown that it is possible to bring directedness to greybox fuzzers at binary level with a very small overhead at both instrumentation-time and run-time. By specializing standard DGF components to UAF, UAFuzz outperforms existing directed fuzzers, both in terms of time to bug exposure and number of successful runs in bug reproduction.

Our final main contribution is the design, implementation and testing of TypeFuzz, which is built on top of UAFuzz. We have shown that our directed techniques proposed in UAFuzz are fruitfully generalized to detecting other typestate bugs, like buffer overflow and NULL pointer dereference. Concretely, TypeFuzz outperforms existing directed fuzzers in several fuzzing evaluation metrics, such as time to bug exposure, number of successful runs and time to reach specific target basic blocks.

Furthermore, both fuzzers UAFuzz and TypeFuzz have been proven effective and efficient in not only bug reproduction, but also in patch testing. Particularly, by leveraging bug traces of disclosed bugs, our directed fuzzers were able to detect different types of unknown vulnerabilities (including incomplete bug fixes) in more recent versions of real-world programs. In summary, the effectiveness and the scalability of our fuzzing frameworks have been validated on various real-world programs to both reproduce disclosed bugs and find new vulnerabilities.

Publications & talks To sum up, our contributions above led to the writing of the following research outputs in security conferences and talks in the PhD Student Symposium of several security workshops in French as follows:

- **Manh-Dung Nguyen**, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre, "*Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities*", The 23nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20), 2020.

- **Manh-Dung Nguyen**, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre, "*About Directed Fuzzing and Use-After-Free: How to Find Complex & Silent Bugs?*", Black Hat USA, 2020.

- **Manh-Dung Nguyen**, "*Directed Fuzzing for Use-After-Free Vulnerabilities Detection*", Rendez-vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI '20 – Session doctorants), 2020.

- **Manh-Dung Nguyen**, "*Directed Fuzzing for Use-After-Free Vulnerabilities Detection*", 19èmes Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL '20 – Session doctorants), 2020.

- **Manh-Dung Nguyen**, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre, "*Binary-level Directed Fuzzing for Complex Vulnerabilities*", *under submission* to IEEE Transactions on Software Engineering (TSE), 2021.

7.1.3 Technical contributions

Furthermore, this thesis comes with several contributions to the open-source community as follows:

- We developed UAFuzz & TypeFuzz [uaf20b], which are new directed greybox fuzzers dedicated to find typestate vulnerabilities, such as UAF or buffer overflows at binary-level;

- We constructed UAF Fuzzing Benchmark [uaf20a], which is a new benchmark including recent bugs found by existing (directed) greybox fuzzers of real-world programs for fuzzing evaluations;

- We contributed to the development of the plugin BinIda and the integration of Graphs of Use-After-Free Extracted from Binary (GUEB) [gue20] into the open-source binary analysis framework Binsec [bin20];

- We reported **39** unknown vulnerabilities of different open-source projects (**17** CVEs were assigned), which were found by our directed fuzzers presented in this manuscript. So far, 30 bugs have been acknowledged and fixed by the developers.

7.2 Perspectives

We present below some promising research directions as extensions of the work discussed in the manuscript.

Hybrid directed fuzzing (*work in progress*) Hybrid directed fuzzing could be more efficient in finding vulnerabilities as software testing techniques, like symbolic execution, static analysis or machine learning, can pinpoint target locations to boost directedness and allow DGF to overcome roadblocks during the search. For example, DRILLERGO [KY19] searches and selects suspicious method call strings in the Control Flow Graph (CFG) in the static analysis phase, and then runs the concolic execution along with path guiding in the backward manner from the target to the start of `main()`. Another natural extension is to combine DGF with latest advances of CGF or other software testing techniques to improve the fuzzing performance in general and also tackle common fuzzing problems, such as magic bytes comparisons or highly-structured file formats.

A hybrid approach, which combines static analysis in GUEB and the directed fuzzer UAFUZZ, could benefit from both sides to detect complex vulnerabilities UAF in binary code *with no prior information*. Our goal is to make the combination between two techniques *fully-automated*, *robust* and *powerful*. First, we improve static analysis techniques proposed in [FMP14,FMB+16] so that it could work with real-world programs in the UAF fuzzing benchmark (in Chapter 4) and produce high-quality static reports. Second, we propose *binary-aware heuristics* combining three (3) different types of derived binary metrics including *complexity, UAF-oriented* and *fuzzing-oriented metrics* to select the most suspicious static reports as targets of our directed fuzzer UAFUZZ. Finally, at runtime, the static analyzer and the fuzzer can communicate to each other to exchange important information, for example target static reports or fuzzing status.

Parallel directed fuzzing In the OSS-FUZZ project [oss20a], Google has been continuously using more than 25,000 machines for fuzzing since 2016 and has found a thousand of bugs in its own software and open source projects. However, there is a very few research work on how to effectively use the hardware resources for fuzzing in parallel (e.g., information synchronization or task division mechanism) to minimize the overlap and maximize the code coverage of all fuzzing instances. Directed fuzzing is very suitable in this setting as each instance will focus only on its assigned task. However, a remaining challenge is to develop directed fuzzing in a "*dynamic*" way such that we can continuously update the graphs to make them more complete and calculate the seed metrics (e.g., distances) on the fly during the fuzzing process.

Directed fuzzing for exploitable vulnerabilities Fuzzing has been proven to be effective in finding a huge number of bugs, but only few of them are exploitable. The exploitability of heap-based vulnerabilities like UAF could be an attractive research direction as developers usually pay more attention on fixing exploitable vulnerabilities first. The goal is to extend the triage step to find potentially exploitable inputs among the crashing ones and then develop a solution to automatically generate exploits [ACHB11, WZX+18, YKK20].

Human-in-the-loop directed fuzzing Fuzzing is currently easy to install and use, but very difficult to be intervened at runtime, especially for non-expert users. However,

leveraging and integrating the knowledge from developers or testers during the fuzzing campaign without restarting the process (e.g., provide an input to bypass a magic bytes comparison or drive the search towards uncovered suspicious functions) can really boost the fuzzer efficiency. We hope that DGF will soon be integrated in the software development life cycle like OSS-Fuzz [oss20a] of Google and CI Fuzz [cif20] of Code Intelligence and eventually become a good testing practice, like writing unit tests.

Acronyms

AAFD Augmented Adjacent-Function Distance. 1, 28
AFL American Fuzzy Lop. 1
AI Artificial Intelligence. 1
AVM Alternating Variable Method. 1

CFG Control Flow Graph. 1, 10, 29, 86
CG Call Graph. 1, 29
CGC Cyber Grand Challenge. 1
CGF Coverage-guided Greybox Fuzzing. xii, 1, 5, 16–18, 25, 28
CVE Common Vulnerabilities and Exposures. 1
CWE Common Weakness Enumeration. 1, 6

DARPA Defense Advanced Research Projects Agency. 1
DBA Dynamic Bitvector Automata. 1
DBI Dynamic Binary Instrumentation. 1, 18
DBMS Database Management System. 1
DCT Dynamic Calling Tree. xii, xiii, 1, 39, 67, 71, 77, 90, 91
DF Double-Free. 1, 14
DGF Directed Greybox Fuzzing. xii, 1, 5, 11, 17, 23, 25, 28, 85
DSE Dynamic Symbolic Execution. 1, 17, 22
DTA Dynamic Taint Analysis. 1, 22

GUEB Graphs of Use-After-Free Extracted from Binary. 1, 11, 86

LLVM Low-Level Virtual Machine. 1

ML Machine Learning. 1, 17, 22

NVD National Vulnerability Database. 1, 6, 91

PoC Proof-of-Concept. 1, 4, 70, 71, 73
PSO Particle Swarm Optimization. 1
PUT Program Under Test. 1, 4, 15

SA Static Analysis. 1, 4
SBF Search-based Fuzzing. 1, 17
SBST Search-based Software Testing. 1, 16, 17
SE Symbolic Execution. 1, 4, 16, 17
SMT Satisfiability Modulo Theories. 1, 16, 17
SOK Systemization of Knowledge. 1, 18

TIR Triaging Inputs Rate. 1
TTE Time-to-Exposure. 1

UAF Use-After-Free. 1, 5, 6, 14, 29, 33

VSA Value Set Analysis. 1, 89, 90

Bibliography

[ACHB11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[afl20a] Afl. http://lcamtuf.coredump.cx/afl/, 2020.

[afl20b] Afl-dynamorio. https://github.com/vanhauser-thc/afl-dynamorio, 2020.

[afl20c] Afl-dyninst. https://github.com/talos-vulndev/afl-dyninst, 2020.

[afl20d] Afl vulnerability trophy case. http://lcamtuf.coredump.cx/afl/#bugs, 2020.

[afl20e] Aflgo. https://github.com/aflgo/aflgo, 2020.

[afl20f] Aflgo's issues. https://github.com/aflgo/aflgo/issues, 2020.

[afl20g] Aflpin. https://github.com/mothran/aflpin, 2020.

[afl20h] Aflplusplus. https://aflplus.plus/, 2020.

[afl20i] American fuzzy lop - a security-oriented fuzzer. https://github.com/google/AFL, 2020.

[afl20j] Technical "whitepaper" for afl. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2020.

[air20] Spanish air force cargo plane on test flight crashes near seville airport. https://www.theguardian.com/world/2015/may/09/spanish-air-force-cargo-plane-crashes-near-seville-airport/, 2020.

[ASAH] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing.

[ASB+19] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[Bac96] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[BAS+19] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: synthesizing structure while fuzzing. In *USENIX Security Symposium (USENIX Security 19)*, 2019.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[BCR21] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and opportunities. *IEEE Software*, pages 1–9, 2021.

[BDM17] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. In *S&P*, pages 633–651. IEEE Computer Society, 2017.

[ber20] Israeli moon lander suffered engine glitch before crash. https://www.space.com/beresheet-moon-crash-engine-glitch.html, 2020.

[bin20] Binsec. https://binsec.github.io/, 2020.

[Böh19] Marcel Böhme. Assurance in software testing: A roadmap. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '19, 2019.

[BPNR17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017.

[BPR16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977.

[CC18] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 758–772, 2018.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[cgc20] Darpa cgc corpus. http://www.lungetech.com/2017/04/24/cgc-corpus/, 2020.

[CGMN12] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143. ACM, 2012.

[CH11] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 2011.

[cif20] Ci fuzz - ide plugin for testing source code. https://www.code-intelligence.com/developers, 2020.

[CJM+19] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1967–1983, 2019.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS XVI*, pages 265–278, 2011.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[CLC19] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. *arXiv preprint arXiv:1905.12228*, 2019.

[clu20] Clusterfuzz. https://google.github.io/clusterfuzz/, 2020.

[CMW16] Maria Christakis, Peter Müller, and Valentin Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 144–155. ACM, 2016.

[cod20] Grammatech codesonar. https://www.grammatech.com/products/codesonar, 2020.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[cve20a] Cve-2017-15939 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-15939, 2020.

[cve20b] Cve-2018-6952. https://savannah.gnu.org/bugs/index.php?53133, 2020.

[CXL+18] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108. ACM, 2018.

[DB15] Adel Djoudi and Sébastien Bardin. Binsec: Binary code analysis with low-level regions. In *TACAS*, pages 212–217, 2015.

[DBF+16] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA*, pages 36–46. ACM, 2016.

[DBT+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 653–656. IEEE, 2016.

[DF04] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, 2004.

[dfp20] Double free in gnu patch. https://savannah.gnu.org/bugs/index.php?56683, 2020.

[DGHK+16] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 110–121. IEEE, 2016.

[DLRA15] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–29, 2015.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[drm20] Dr.memory. https://www.drmemory.org/, 2020.

[eth20] Should a self-driving car kill the baby or the grandma? depends on where you're from. https://www.technologyreview.com/2018/10/24/139313/a-global-ethics-study-aims-to-help-ai-solve-the-self-driving-trolley-prob 2020.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[fac20] Facebook Paid $22 Million in Bug Bounty Rewards in 2019. https://www.securityweek.com/facebook-paid-22-million-bug-bounty-rewards-2019, 2020.

[FDC19] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. Weizz: Automatic grey-box fuzzing for structured binary formats. *arXiv preprint arXiv:1911.00621*, 2019.

[FGRY03] John Field, Deepak Goyal, G Ramalingam, and Eran Yahav. Typestate verification: Abstraction techniques and complexity results. In *International Static Analysis Symposium*. Springer, 2003.

[FLDGB19] Andrew Fasano, Tim Leek, Brendan Dolan-Gavitt, and Josh Bundt. The rode0day to less-buggy programs. *IEEE Security & Privacy*, 17(6):84–88, 2019.

[FMB+16] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, page 2. ACM, 2016.

[FMEH20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[FMP14] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[FMRS12] Azadeh Farzan, P Madhusudan, Niloofar Razavi, and Francesco Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[fuz20] Fuzzbench - fuzzer benchmarking as a service. https://github.com/google/fuzzbench, 2020.

[FYD+08] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2008.

[GAAH19] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. Antifuzz: Impeding fuzzing audits of binary executables. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1931–1947, 2019.

[gft20] Google fuzzer testsuite. https://github.com/google/fuzzer-test-suite, 2020.

[gnu20] Gnu patch. https://savannah.gnu.org/projects/patch/, 2020.

[goo20] Google has paid security researchers over $21 million for bug bounties, $6.5 million in 2019 alone. https://venturebeat.com/2020/01/28/google-has-paid-security-researchers-over-21-million-for-bug-bounties-6-5- 2020.

[GPS17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

[gpz20] Sockpuppet: A walkthrough of a kernel exploit for ios 12.4. https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html, 2020.

[gue20] Gueb: Static analyzer detecting use-after-free on binary. https://github.com/montyly/gueb, 2020.

[GZC+20] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA. https://www. usenix. org/conference/usenixsecurity20/presentation/gan*, 2020.

[GZQ+18] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 660–677, 2018.

[hac20] Hackers Earn Record-Breaking $100 Million on HackerOne. https://www.businesswire.com/news/home/20200527005320/en/Hackers-Earn-Record-Breaking-100-Million-on-HackerOne, 2020.

[Ham02] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 2002.

[hea20] The heartbleed bug. https://heartbleed.com/, 2020.

[HHP20] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[HP07] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007.

[HSNB13] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *Proceedings of the 22nd USENIX Security Symposium*, pages 49–64, 2013.

[HYW+20] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. 2020.

[ida20] Ida pro. https://www.hex-rays.com/products/ida/, 2020.

[JC19] Leonid Joffe and David Clark. Directing a search towards execution properties with a learned fitness function. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 206–216. IEEE, 2019.

[JHS+19] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-fuzzing techniques. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1913–1930, 2019.

[JO12] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 474–484. IEEE, 2012.

[JRGB18] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. Tiff: using input type inference to improve fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 505–517, 2018.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[KT14] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[KY19] Juhwan Kim and Joobeom Yun. Poster: Directed hybrid fuzzing on binary code. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2637–2639, 2019.

[laf20] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2020.

[LCC+17] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 2005.

[LHG13] Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: An open source tool for search based software testing of c programs. *Information and Software Technology*, 55(1):112–125, 2013.

[lib20a] Libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2020.

[lib20b] libfuzzer tutorial – heartbleed. https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md#heartbleed, 2020.

[LJC+18] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 809–814, 2018.

[LJZ+19] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1949–1966, 2019.

[LPSS18] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

[LS18] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485. ACM, 2018.

[LSJ+15] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[LXC+19] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang 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*, pages 533–544. ACM, 2019.

[LZY+19] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 249–259. IEEE Press, 2019.

[MC13] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[McM11] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[MCY+18] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, Baltimore, MD, 2018. USENIX Association.

[MFS90] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[MHH+19] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[Mic20] Microsoft. Project springfield. https://www.microsoft.com/en-us/security-risk-detection/, 2020.

[MLW09] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[muz20] MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multi-threaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[nas20] Mars 2020 perseverance rover - nasa mars. https://mars.nasa.gov/mars2020/, 2020.

[NIS20] NIST. Juliet test suite for c/c++. https://samate.nist.gov/SARD/testsuite.php, 2020.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan*, volume 42-6, pages 89–100. ACM, 2007.

[nvd20] Us national vulnerability database. https://nvd.nist.gov/vuln/search, 2020.

[OHLP14] Mads Chr Olesen, René Rydhof Hansen, Julia L Lawall, and Nicolas Palix. Coccinelle: tool support for automated cert c secure coding standard certification. *Science of Computer Programming*, 91:141–160, 2014.

[OHPP18] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1475–1482, 2018.

[one20] Microsoft announces new project onefuzz framework, an open source developer tool to find and fix bugs at scale. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/, 2020.

[ÖRBG20] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[oss20a] OSS-Fuzz: Continuous Fuzzing Framework for Open-Source Projects. https://github.com/google/oss-fuzz/, 2020.

[oss20b] Oss-fuzz: Five months later, and rewarding projects. https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2020.

[PBS+19] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[pea20] Peach fuzzer. https://www.peach.tech/, 2020.

[PF20] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 181–198, 2020.

[PLL+19] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 605–616. IEEE, 2019.

[PNRR15] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 891–901. IEEE, 2015.

[PSP18] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[PTS+17] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632. IEEE, 2017.

[PZKJ17] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2155–2168, 2017.

[r220] Radare2: Libre and portable reverse engineering framework. https://rada.re/n/, 2020.

[RBB+19] Frédéric Recoules, Sébastien Bardin, Richard Bonichon Bonichon, Laurent Mounier, and Marie-Laure Potet. Get rid of inline assembly through verification-oriented lifting. In *ASE*. IEEE, 2019.

[RBS17] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[rev20] Detection deficit: A year in review of 0-days used in-the-wild in 2019. https://googleprojectzero.blogspot.com/2020/07/detection-deficit-year-in-review-of-0.html, 2020.

[RJK+17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[rod20] Rode0day. https://rode0day.mit.edu/, 2020.

[RPDGH18] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 224–234. ACM, 2018.

[SAA+17] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-cube: High-dimensional hypervisor fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[SGS+16] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[SLR+19] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[SPE+18] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[SRDK19] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.

[SW13] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, pages 25–59, 2013.

[SWD+17] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 347–362, 2017.

[SX16] Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 460–473. ACM, 2016.

[SY86] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 1986.

[Sze17] László Szekeres. *Memory corruption mitigation via hardening and testing*. PhD thesis, Ph. D. Dissertation. Stony Brook University, 2017.

[uaf20a] Uaf fuzzing benchmark. https://github.com/strongcourage/uafbench, 2020.

[uaf20b] Uafuzz. https://github.com/strongcourage/uafuzz, 2020.

[VD00] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[WBS01] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and software technology*, 43(14):841–854, 2001.

[WCWL17] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[WJL+] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization.

[WJLL19] Yan Wang, Peng Jia, Luping Liu, and Jiayong Liu. A systematic review of fuzzing based on machine learning techniques. *arXiv preprint arXiv:1908.01262*, 2019.

[WSZ16] Weiguang Wang, Hao Sun, and Qingkai Zeng. Seededfuzz: Selecting and generating seeds for directed fuzzing. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 49–56. IEEE, 2016.

[WWL+20] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *42nd International Conference on Software Engineering*. ACM, 2020.

[WWLZ09] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

[WXL+20] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *42nd International Conference on Software Engineering*, 2020.

[WZ20] Pengfei Wang and Xu Zhou. Sok: The progress, challenges, and perspectives of directed greybox fuzzing. *arXiv preprint arXiv:2005.11907*, 2020.

[WZX+18] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1914–1927. ACM, 2018.

[XKMK17] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328, 2017.

[YKK20] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[YLX+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[You15] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[YSCX17] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 42–54, New York, NY, USA, 2017. ACM.

[YSCX18] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337. IEEE, 2018.

[YWM+19] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[YZC+17] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[ZCH+20] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. *arXiv preprint arXiv:2006.02398*, 2020.

[ZDYX19] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

[ZLW+] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning.

[ZWL+19] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, Chengnian Sun, and Yu Jiang. Visfuzz: understanding and intervening fuzzing with interactive visualization. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1078–1081. IEEE, 2019.