RUB

Dissertation

Automated Program Analysis:

Reproducibility, Flexibility Requirements, and Risks

Moritz Schloegel

A dissertation submitted to the Faculty of Computer Science

at Ruhr University Bochum for the degree of

Doctor of Natural Sciences (Dr. rer. nat.)

Reviewed by: Prof. Dr. Thorsten Holz
CISPA Helmholtz Center for Information Security

Prof. Dr. Christopher Kruegel
University of California, Santa Barbara

Dr. Marcel Böhme
Max Planck Institute for Security and Privacy

Defended on: 7th of May, 2024

Abstract

Software is increasingly integrated into our daily routines and processes, powering everything from simple entertainment to critical infrastructure. With countless developers writing thousands of lines of code every hour, manually ensuring that software is bug-free is impossible. Automating program analysis is our best chance of securing as much of this vast software landscape as possible. Despite manifold techniques being readily available to analyze, test, and verify software, we have not yet found a panacea. Problems include the misalignment between results reported in some papers and those observed in practice, the requirement for human intervention in non-standard use cases, and the risk of equipping malicious actors with more powerful tools.

This thesis advances the state of the art threefold: First, we conduct a thorough literature analysis of automated software testing techniques in the field of fuzzing, analyzing whether the results of the proposed fuzzers are reproducible. Such reproducibility is vital to enable follow-up research and facilitate industry adoption. Based on our insights, we provide future work with recommendations for evaluating fuzzers. Second, we observe that state-of-the-art automated exploit generation approaches are inflexible and only work within a narrow set of common tasks. We propose a new approach with unprecedented flexibility toward generating so-called gadget chains that allow us to model arbitrary constraints. At the same time, our technique can prove that no such chain can exist for the given conditions. This can help developers assess the exploitability of reported bugs, supporting them with prioritizing critical vulnerabilities. In the third and last contribution, we study how advancing to more powerful automated analyses hurts software protection. Code obfuscation protecting intellectual property, for example, in DRM systems, relies on security-by-obscurity, usually achieved through complexity. We find that current automated program analysis techniques are highly effective in stripping away this protection, even for commercial state-of-the-art obfuscators. To step up protection, we show how a combination of obfuscation techniques relying on inherent weaknesses of the analysis methods can achieve resilience against automated deobfuscation attacks. This work underlines how improving automated analyses in one domain may pose risks to another, calling for research to exercise prudence.

Acknowledgements

A good PhD is like a roller coaster in the dark: Unpredictable turns ensue in quick succession, there's too many ups and downs to keep track of, but in the end you had a great time. And looking back you can nothing but wonder how time flew by and how you made it to the end. My ride, albeit wilder than I imagined, was an absolute blast and I'm humbled to have worked with such incredibly bright and awesome people. Trying to distill this experience into mere words and do everybody and their profound impact on my journey justice is a futile endeavor – one I'll attempt regardless; after all, what research has taught me is that sometimes you just have to bang your head against the wall often enough to make it work.

Thorsten, thank you for these amazing years: Without you, this journey would not have been possible, and I appreciate you giving us all the leeway to make this such a fun experience. Our discussions and your thoughtful feedback will be sorely missed. Chris, Marcel thank you for taking your precious time to review this thesis – I'm looking forward to many future collaborations. Syssec family, particularly Cornelius, Emre, Joel, Johannes, Lea, Lukas, Merlin, Nico, Nils, Tim, Thorsten, and Tobias, I owe you: I've not only learned a lot from you, but you always had my back, made the ride the fun it was, and didn't mind all the banter! I'm happy to count you among my friends. To the many more fine folks that I've met during this journey—to many to name them all—a heartfelt thank you. It is the daily interactions and fun conversations that fuel our research. May our paths cross many more times in future.

Behind the curtain, there has been an unwavering support by my family, friends and girlfriend. Not only did you put me onto the right trajectory, but you kept me on track throughout the whole journey. Whenever a looming deadline limited my time, you made up for it. You emphatically shared all the losses and loudly celebrated the wins. I could not have asked for more support. To my family and friends, thank you for being there whenever I needed you. To Claudia, thank you for your endless love, your patience when I stayed late at work again, and the many more things.

Contents

I Introduction 1

1 Introduction 3

2 Technical Background 11
- 2.1 Fuzzing 12
  - 2.1.1 Blind Fuzzers 13
  - 2.1.2 Feedback-driven Fuzzers 13
  - 2.1.3 Heavyweight Feedback-driven Fuzzers 16
  - 2.1.4 Input Generation 16
- 2.2 Exploitation 17
  - 2.2.1 Memory Corruption 17
  - 2.2.2 Code Injection 18
  - 2.2.3 Code Reuse 19
  - 2.2.4 Data-only Attacks 19
  - 2.2.5 Mitigations 20
- 2.3 Obfuscation 21
  - 2.3.1 MATE Threat Model 21
  - 2.3.2 Obfuscation Techniques 22
  - 2.3.3 Automated Deobfuscation Attacks 23

3 Reproducibility, Flexibility Requirements, and Risks 27
- 3.1 Reproducibility of Fuzzing 27
  - 3.1.1 Key Results 30
  - 3.1.2 Discussion 36
  - 3.1.3 Limitations 38
  - 3.1.4 Future work 39

3.2 Flexibility for Automated Exploit Generation 40
3.2.1 Key Results 41
3.2.2 Discussion 42
3.2.3 Limitations 43
3.2.4 Future Work 44
3.3 Risks of Automation: Weakening Obfuscation 45
3.3.1 Key Results 47
3.3.2 Discussion 48
3.3.3 Limitations 50
3.3.4 Future Work 51

4 Conclusions **53**

References **57**

II Publications 79

List of Publications **81**

A SoK: Prudent Evaluation Practices for Fuzzing **85**
1 Introduction 89
2 Fuzzing Evaluation Guidelines 91
2.1 Background on Fuzzing 91
2.2 Guidelines of *Evaluating Fuzz Testing* 92
2.3 Guidelines of *FuzzBench* 93
2.4 Guidelines of *On the Reliability of Coverage* 93
2.5 Fuzzing Benchmarks 93
3 Literature Analysis 94
3.1 Method 94
3.2 Results 96
4 Artifact Evaluation 107
4.1 Case Study: Artificial Runtime Environment and Unique Crashes 108
4.2 Case Study: Exaggerated Vulnerabilities 109
4.3 Case Study: Missing Baselines 110
4.4 Case Study: Non-reproducible Measurements 113

4.5 Case Study: Uncommon Metrics . . . 114
4.6 Case Study: Unclear Documentation . . . 116
4.7 Case Study: Incomplete Artifact . . . 117
4.8 Case Study: Unfair Coverage Measurements . . . 118
5 Revised Best Practices for Evaluation . . . 119
5.1 Reproducible Artifact . . . 119
5.2 Targets under Test . . . 120
5.3 Comparison to Other Fuzzers . . . 121
5.4 Evaluation Setup . . . 121
5.5 Evaluation Metrics . . . 122
5.6 Statistical Evaluation . . . 123
6 Conclusion . . . 124

B Automating Code-Reuse Attacks Using Synthesized Gadget Chains 147
1 Introduction . . . 150
2 Shortcomings of State-of-the-Art Approaches . . . 152
3 Design . . . 153
3.1 Gadgets . . . 154
3.2 Logical Encoding . . . 154
3.3 Preconditions and Postconditions . . . 158
3.4 Formula Generation . . . 159
3.5 Algorithm Configuration . . . 160
4 Implementation . . . 160
5 Evaluation . . . 161
5.1 Setup . . . 161
5.2 Finding a Chain . . . 162
5.3 Real-World Applicability . . . 164
5.4 Target-Specific Constraints . . . 165
5.5 Chain Statistics . . . 166
5.6 `SGC`'s Configuration . . . 167
6 Discussion . . . 168
7 Related Work . . . 169
8 Conclusion . . . 170
A Memory Modeling . . . 174

B `dnsmasq` CVE-2017-14493 175

C Hardening Code Obfuscation Against Automated Attacks 179

1 Introduction 182
2 Technical Background 185
2.1 VM-based Obfuscation 185
2.2 Mixed Boolean-Arithmetic 187
3 Automated Deobfuscation Attacks 187
4 Design 190
4.1 Design Principles 190
4.2 Attacker Model 192
4.3 Key Selection Diversification 193
4.4 Syntactic Complexity: MBA Synthesis 194
4.5 Semantic Complexity: Superoperators 196
4.6 Synergy Effects 198
4.7 Verification of Code Transformations 198
5 Implementation 199
6 Experimental Evaluation 200
6.1 Benchmarking 201
6.2 Resilience 203
6.3 Evaluation of Key Encodings 204
7 Discussion 215
8 Related Work 216
9 Conclusion 217

Part I

Introduction

Introduction

Software is a ubiquitous cornerstone of our modern, digital lives, connecting us with friends, peers, and the world. Its ever-growing role in our daily lives and reliance on its continued availability and correct function make securing software an imperative goal. With the vast diversity of available hardware, software, and technologies, manually ensuring the safety and security of all systems and applications in this ecosystem is no longer feasible. There are dedicated teams and even companies providing code audits as their sole service, yet the number of employees writing new code or changing existing code easily outpaces them. The most promising solutions to address this imbalance are to enforce security by design and to automate software testing. The former proactively limits the attack surface and provides significantly stronger security guarantees than the latter. As of today, security by design is also elusive. For many different reasons, security is often still an afterthought when planning and implementing software projects. One cause are wrong incentives, where security breaches are priced into the product rather than spending the additional money on security beforehand. Worsening the situation, it is difficult to measure security, which leads to scenarios where a cheaper yet potentially insecure design is more appealing to many companies. In general, the multitude of companies and individuals writing code for different purposes renders it unlikely that humanity will arrive at a widespread mentality of security by design within the foreseeable future. However, even when assuming that any code written from tomorrow on is secure by design, our huge legacy codebases still expose ample attack surfaces. Rewriting all existing code is hardly attractive due to the associated costs, the risk of introducing new errors, or simply for fear of losing compatibility with

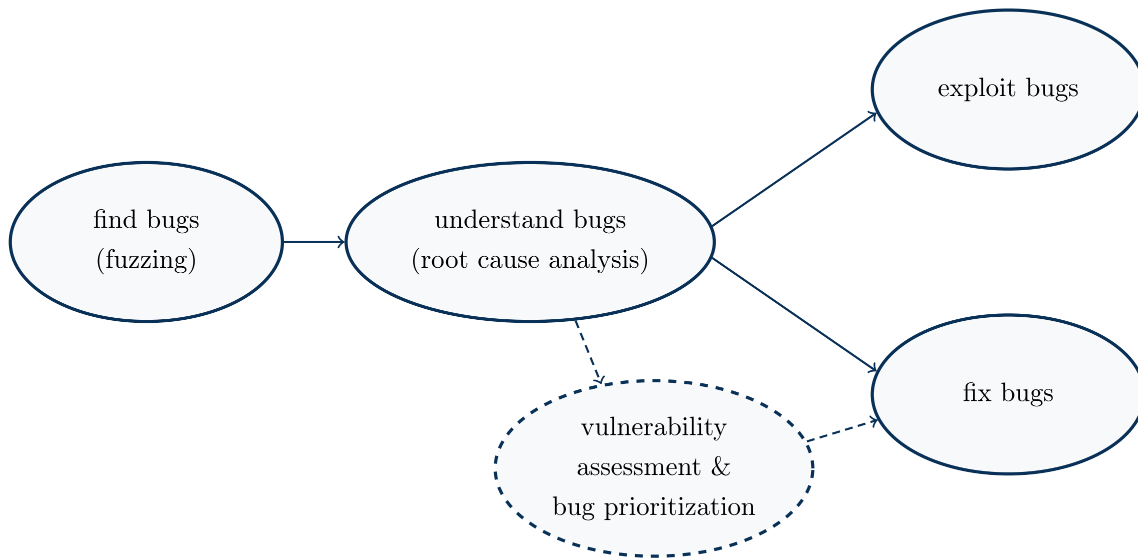


Figure 1.1: Multi-step pipeline of (1) finding bugs, then (2) understanding them and their actual root cause. Once we have sufficient knowledge, we can (3) either derive an exploit or patch the underlying bug. When dealing with many bug reports, assessing and ranking them by their criticality can help prioritizing more severe bugs.

other software. That said, we emphasize that pushing towards security by design—to ensure future code is more secure than existing one—is a worthwhile goal, but it is not a panacea. Compared to this concept of security by default, automated testing appears pale: It can only find some bugs but never guarantee the absence of bugs. Still, absent security-by-design, it is our best attempt at uncovering as many programming errors as possible and steadily raising the cost for a malicious actor by fixing at least the easy-to-find, “low-hanging fruits” of the bugs. Even though many different forms of testing exist, not all are well-suited for our goal of securing as much code as possible. For example, testing programs manually fails to address the sheer amount of code that exists and is produced every day. Our best and only realistic chance at scaling testing to more software than a handful of selected ones is the *automation of program analyses*.

When breaking down this problem of scaling and automating the testing process, we quickly appreciate that this is a complex problem consisting of multiple steps [169]. We can consider this testing process as a *pipeline* with three distinct steps (see Figure 1.1): First, we need to *find bugs*, for example, by generating some input that causes the target program to crash. Then, we need to *understand the bug*, the capabilities it provides an attacker with, or even its root cause. In some cases, the root cause and bug may be

synonymous, while millions of instructions may separate them in others [24]. Once we have a sufficient understanding of the bug, we can either attempt to exploit it or try to derive a patch to fix the issue. When fixing the bug is not an option, for example, due to the lack of access to the source code, we can still attempt to assess the bug's exploitability and help with prioritizing critical bugs. Prioritization may be particularly important when dealing with a large influx of bugs, which we can often observe when a program is fuzzed for the first time [72].

Even when focusing on a single step in this pipeline of observing a crash, understanding the underlying bug, and working towards an appropriate fix (or exploit), it is challenging to come up with a single generic solution. In fact, much research has been conducted on individual steps without finding a conclusive solution yet. While all this research belongs to the field of *program analysis* [131], we can make more fine-granular distinctions, such as between *static* techniques that reason on all possible program behaviors and *dynamic* ones that work on concrete execution traces. For finding bugs, dynamic techniques have proven highly effective: Even though they cannot reason on all possible program behavior and, thus, never prove the absence of bugs, they have an excellent track record in finding bugs based on concrete program input [72], thereby avoiding false positives which plague static analysis [100, 110, 125]. A key driver behind this immense success is *fuzzing*, a comparably straightforward approach that executes the system under test in quick succession with different inputs, slightly mutating these inputs between executions, and tracking the execution path of each one. Receiving information on the covered code as feedback, the fuzzer can make informed decisions on which inputs have observed new parts of the program and steer the execution towards novel, unseen behavior. By creating many different inputs and not adhering to implicit assumptions programmers may have encoded in their programs, fuzzing has turned out to be effective in creating such inputs that cause the program under test to crash. In short, fuzzing is an automated bug-finding technique that produces inputs crashing the program [113, 202]. Such an input proves the existence of some bug within the program but may not necessarily reveal the root cause or other details. In some cases, more information can be gained using sanitizers [166, 175] that instrument the program and abort execution on typical memory errors, such as out-of-bounds reads, use-after-frees, or reading uninitialized memory. In other cases, dedicated approaches are needed to understand the root cause of the underlying bug [24, 139, 199, 200] or the capabilities it provides an attacker with [97, 203]. Now, an impressive number of

new fuzzing techniques has been proposed over the years [11, 18, 22, 26, 27, 32, 33, 60, 61, 63, 82, 103, 112, 113, 127, 143, 155, 176, 178, 201, 202]. With this large amount of available techniques, industry practitioners and fellow researchers have a hard time keeping up with the field. The former may want to find a fuzzer that works for their particular use case, while the latter strive to find problems and oversights of existing techniques to improve the state of the art further. When looking at current fuzzing papers, they fall short of providing the necessary requirements for either case: A paper may not cover a particular use case, and comparing different paper evaluations against each other is surprisingly challenging, with different setups, targets, or experiments making the comparison of reported performance numbers usually pointless. In both cases, the only way forward is *reproducibility*: A reproducible fuzzer features sufficient documentation and is open-source, allowing industry practitioners to simply test it for their individual use case. Similarly, a reproducible fuzzer enables other researchers to confirm results, build upon a technique, and properly evaluate against it, creating a solid foundation for future research and advancing the state of the art. Despite the large amount of proposed techniques, the level of reproducibility remains unknown.

To change this, we study the reproducibility of fuzzing evaluations for 150 papers and attempt to reproduce the results of eight fuzzing artifacts. Based on the insights thereof, we then outline revised guidelines to support future work with conducting reproducible evaluations. We believe reproducibility is vital for a sustainable and reliable automation of bug finding.

When turning our focus from the first to the last step of the bug-finding pipeline, we find comparably fewer research works studying automated exploit generation [13, 31, 86, 165, 191, 206]. Essentially, these tools' task is to turn a crashing input into one that meets some attacker goal. Depending on the underlying bug, some focus on manipulating the heap layout [87, 88, 191, 205], while others study how to piece together small snippets of code, called *gadgets* in the context of such code reuse attacks. By chaining these gadgets together, an attacker can implement arbitrary computations [167]. This type of approach has a long history in the context of exploitation [164, 167, 172]; however, automatic state-of-the-art tools [7, 121, 152, 157] often rely on inflexible heuristics to find and chain gadgets. Their inflexibility forbids their use under circumstances where specific conditions must be met. One example could be a DRM system that computes a checksum over memory contents in which the gadget chain will be placed. Encoding such atypical constraints is difficult in current tools, restricting their use to

more common scenarios or requiring a human expert to intervene. Another noteworthy property of most research in the domain of automated exploit generation is their strict focus on the offensive side, i. e., how to automatically generate an exploit for the sake of exploitation, with little consideration for potential applications in the realms of assessing and prioritizing bugs. However, with the state of the art in automated program repair [68, 124] being unable to replace human intervention reliably [20, 116, 207], one promising avenue towards at least supporting the developer tasked with fixing a bug is using such automated exploit generation systems to verify whether the bug is exploitable. If so, the underlying bug requires urgent intervention to address security concerns. If this is not the case, ideally, we would like to have some guarantees on whether the bug can be exploited at all. While such guarantees are challenging to provide in the general case, current tools provide no information other than failing to construct an exploit, reducing their appeal to developers.

To address these two shortcomings, this work proposes a new approach that uses a logical encoding of gadgets to transfer the task of finding a gadget chain to finding a satisfying assignment of variables, a task where SMT solvers excel. This way, we can easily encode arbitrary constraints, leading to unprecedented flexibility, and, importantly, the SMT solver can prove that no gadget chain exists for a specific set of pre- and postconditions.

With this discussion of how automating program analysis can help us find bugs and automatically generate exploits, we have only considered the positive aspects of automation. Yet, strengthening automated analyses leads to stronger automated attacks that can be used to defeat software protection. When looking at intellectual property in the form of secret algorithms, Digital Rights Management (DRM) systems, or anti-cheat engines all depend on the notion of secrecy, complexity, and a general opaqueness of underlying code and data. They protect the inner workings of their code by making it hard to extract, analyze, and understand, even in scenarios where an attacker has complete control over the execution and its environment. Amongst others, they rely on *code obfuscation* to achieve protection. Despite the industry's heavy reliance on obfuscation [48, 52, 136, 173, 187], we observe that commercial state-of-the-art obfuscators can be significantly simplified using *automated analyses*. Here, program analysis is used to remove complexity and undo obfuscation, akin to how compilers optimize programs. More powerful analyses risk intellectual property theft and enable adversarial actors

to overcome DRM and anti-cheat systems, undermining the security guarantees that the industry relies on.

To study the real impact of automated analyses against obfuscated code, we first quantify their effectiveness against state of the art. This work then proposes a new, hardened technique that combines multiple obfuscation techniques to achieve lasting resilience against automated deobfuscation attacks. By relying on inherent weaknesses of the analysis techniques, we show that automation for adversarial purposes can be stopped. This switch from the field of finding bugs to software protection highlights that improving automation can also lead to new risks and downsides that research needs to address.

Thesis contributions. In summary, we make the following key contributions:

- *Reproducibility of fuzzing.* We study the *reproducibility* of automated bug-finding approaches in the form of fuzzing. To do so, we analyze 150 scientific publications on A* conferences, study their setup, outline potential pitfalls, and attempt to reproduce eight evaluations empirically. Based on our findings, we make recommendations on how to improve the reproducibility of future fuzzing evaluations.
- *Flexible automated exploit generation.* Complementing existing approaches in the realm of automated exploit generation, we propose a novel technique that features unprecedented flexibility, allowing us to model all constraints that can be expressed as logical formulas. At the same time, this technique can prove that no gadget chain can exist for the given pre- and postconditions, helping to assess exploitability.
- *Hardening against automated deobfuscation.* Highlighting the downsides of more powerful automated program analyses, we turn to software protection and obfuscation in particular. We show that automated analyses can simplify state-of-the-art commercial obfuscators, and we propose several techniques that, in combination, achieve lasting resilience against automated deobfuscation attacks.

Thesis structure. This work is a cumulative dissertation based on three peer-reviewed papers, and it is divided into two parts: The first one provides an introduction to relevant topics, in particular fuzzing, automated exploit generation, and code obfuscation, and outlines the broader research context. It is split into the introduction, followed by a technical background, before focusing on this thesis' contributions to automate program analyses. The second part of this dissertation then provides the verbatim publications underpinning this thesis in Appendices A to C. Their publication data is as follows.

[A] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. "SoK: Prudent Evaluation Practices for Fuzzing". In: *IEEE Symposium on Security and Privacy (S&P)*. 2024

[B] Moritz Schloegel, Tim Blazytko, Julius Basler, Fabian Hemmer, and Thorsten Holz. "Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains". In: *European Symposium on Research in Computer Security (ESORICS)*. 2021.

[C] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. "Loki: Hardening Code Obfuscation Against Automated Attacks". In: *USENIX Security Symposium*. 2022.

Technical Background

This thesis inherently touches on many topics across systems security research in general and program analysis in particular. The following section provides a brief introduction to key concepts and techniques that this work is based on.

Program analysis. Broadly speaking, any technique or method used to observe, study, and understand the behavior of programs is referred to as program analysis [131, 204]. Its goals can vary widely and heavily depend on the respective context. They can be as small as determining a single program property or as large as ensuring the absence of bugs for an entire program. This thesis focuses on two distinct scenarios, namely finding bugs (thus ensuring correctness) and identifying a simpler yet equal representation of a program to "undo" obfuscation (akin to optimizing it). As many means exist to achieve these goals, we can divide them based on whether they execute the program (dynamic analysis) or not (static analysis). Throughout this thesis, we will see techniques from both domains, however, with a focus on dynamic ones. Particularly for bug finding, dynamic techniques have proven effective while fielding few false positives [82]. For software protection, where many techniques are used to make a program as complex as possible, dynamic techniques usually offer better insights, as they can trace the concrete execution path. Even though we consider various static approaches, this thesis does not belong to the domain of approaches strongly grounded in math. For a more formal yet excellent introduction, interested readers may want to refer to Nielson et al.'s *Principles of Program Analysis* [131] and Zeller's hierarchy as a good overview [204]. In the following, we briefly discuss the domains relevant to this work: fuzzing, exploitation, and code obfuscation.

2.1 Fuzzing

Fuzzing, also called *fuzz testing*, is a highly effective, dynamic bug-finding technique [72]. At its core, it generates many different inputs and executes a system under test with these inputs, monitoring for unexpected or erroneous behavior, such as program crashes.

Despite many recent advancements, the origins of fuzzing date back almost 40 years. In the fall of 1988, Barton Miller launched a project to test the reliability of OS utility programs, proposing that students write a *fuzz generator* producing random inputs to break these programs. Two years later, Miller et al. published the results of their study [122]: Out of 88 programs tested across six operating systems, 42% (37) were found to cause a crash or hang on at least one of the tested operating systems. On average, 24% to 33% of programs were found to crash on an operating system. This number is astonishingly high, especially considering their simple approach that fed random inputs to the programs. While this early fuzzing lacks many of the advanced features common in modern fuzzers, it already employs the notion of bug oracles monitoring for program *crashes* or *hangs*, which are still the primary error detection techniques used today.

Even though this early success proved its practical viability, fuzzing has received little attention from academia and industry, primarily due to its limited, brute-force-esque nature. Then, in 2013, Michał Zalewski made a groundbreaking change with his fuzzer AFL [202] that relies on lightweight coverage feedback. Here, the target program is instrumented and reports the edges taken by a specific input back to the fuzzer. Knowing what parts of the program were covered by one input, the fuzzer can now judge whether this particular input helped it explore new, unseen program behavior and, thus, is worthy of being kept for future mutation. This efficient and effective form of guidance pushed fuzzing back into the spotlight of academia, and many research works aimed at optimizing different aspects of fuzzing ensued [61, 113].

The success of fuzzing is due to many factors, most notably the automated fashion it operates in, allowing it to achieve high throughput in terms of executed inputs, its capability to effectively mutate inputs such that deeper program parts are tested, and its ability to use bug oracles to identify triggered faults—all without human intervention. Complemented by a good fuzzing harness that allows to explore the system under test effectively, a fuzzer can quickly identify many faults.

Nowadays, many different fuzzing techniques exist that focus on improving different aspects of fuzzing [113]. To better highlight the diversity of proposed techniques, we differentiate fuzzers based on two (somewhat interlaced) criteria in the following: The degree and sophistication of feedback they receive and the knowledge of the input structure.

2.1.1 Blind Fuzzers

Sometimes also referred to as *black-box fuzzers*, this type of fuzzer is close to the original fuzz generator proposed by Miller et al. [122]. It receives no feedback from within the system under test; in particular, no *coverage feedback*, i. e., information on what path an input has taken in the target, is available. While comparably simple to implement, the absence of information leads to situations where the fuzzer has produced an input solving some complex constraint in the target program without recognizing it did so. Consequently, it might simply discard the valuable input and generate another one, constantly having to produce by chance such inputs that reach deeper parts of the target program. Overall, this type of fuzzer struggles when testing programs that impose many or complex constraints, as the fuzzer is unlikely to continuously generate inputs that solve these constraints. At the same time, the lack of instrumentation enables a high throughput, allowing the fuzzer to test many different inputs. Notably, blind fuzzing may be the last resort in some scenarios, such as fuzzing on hardware, where no instrumentation can be inserted and, thus, it is difficult to extract feedback [32, 58, 156].

2.1.2 Feedback-driven Fuzzers

The most prominent category of fuzzers today, *feedback-driven* fuzzers (also called *greybox fuzzers*) rely on some information extracted from the system under test to guide their exploration and testing process. Dating back to AFL [202], the most popular feedback is *code coverage feedback*, where the fuzzer receives information on the edges a specific input has taken in the system under test. Other feedback can comprise, for example, data flow [89]. The fundamental advantage of feedback is that the fuzzer can distinguish whether a newly generated input exercised new, interesting parts in the program or whether it merely explored the same parts again. Based on this information, the fuzzer then can decide whether to keep a particular input for further mutation or

discard it. This cycle of testing an input, receiving feedback on how it performed, and then discarding or mutating it further is referred to as the *fuzzing loop*, which is continuously repeated throughout a fuzzing campaign.

To maintain a high throughput, it is crucial that the inserted instrumentation is lightweight and not slowing down the execution of the target.

Binary-only fuzzing. In the absence of source code, inserting the instrumentation needed for collecting and reporting feedback at compile-time is impossible. Instead, various approaches have been devised to enable fuzzing to profit from the advantages of coverage feedback even without access to the source code. First, *Dynamic Binary Instrumentation (DBI)*, such as Intel PIN [111] or DynamoRIO [73], can insert hooks that report coverage at runtime. However,, dynamically instrumenting binaries comes with a significant performance overhead [127]. Moving the runtime overhead ahead of the execution, static binary instrumentation or binary rewriting, e. g., as done by Dyninst [181], RetroWrite [49], or Zipr [84, 91], offers another opportunity to insert instrumentation after compilation. Researchers, especially Nagy et al., have made various efforts to further increase its efficiency by introducing coverage-guided tracing [126, 128], i. e., only tracing the coverage of inputs that exercised new coverage, and porting compiler optimization techniques to their rewriting efforts [127]. Instead of instrumenting the program, hardware features such as Intel PT allow tracing the program execution [160, 161, 178]. While hardware-assisted approaches exhibit good performance characteristics, they are only available on systems featuring the particular hardware feature.

Challenges in fuzzing. Feedback-driven fuzzers show a remarkable capability of efficiently exploring the code of many programs. Nonetheless, some programs resist in-depth testing, with two remarkable challenges: *fuzzing roadblocks* and *complex input formats*. The former refers to all code constructs imposing constraints that are difficult to solve for a fuzzer. For example, checksums or magic values [11] are one notorious source of blockade, as the fuzzer has to correctly guess one particular value within a large search space to solve the underlying comparison. With this being unlikely, a naive fuzzer will rarely visit the branch guarded by this comparison, limiting its potential for exploration. This type of fuzzing roadblock sparked the creation of a whole new category of fuzzers that rely on more complex program analysis techniques,

[70, 96, 143, 176, 201] or taint tracking [33, 67, 145, 190], to solve these challenges. As an alternative with better performance, research proposed lightweight optimizations that help the fuzzer find the correct value. Among them, LAF-Intel [182] or *Compare Coverage* [75] (`cmpcov`), compiler passes that convert larger comparisons into multiple smaller ones. For example, they replace a single 64-bit comparison with four 8-bit comparisons. This way, the fuzzer has to guess only a single byte correctly at a time, which is significantly easier. Guessing the correct value unlocks new coverage, incentivizing the fuzzer to store this input before turning towards solving the next byte. Another lightweight technique to tackle such fuzzing roadblocks is *input-to-state correspondence*, introduced by RedQueen [11] and later called `cmplog` [3]. Its underlying insight is that input to a program often undergoes few (if any) transformations before being used in a comparison. Consequently, Aschermann et al. propose to extract the values that the input bytes are compared to and then set the corresponding bytes in the fuzzer-controlled input to match the program's expectation [11]. A third technique to mitigate the impact of these fuzzing roadblocks is *Compare Coverage (cmpcov)* [75], which provides the fuzzer with a hamming distance of how close that provides sub-instruction information akin in spirit to LAF-Intel and thereby allows the fuzzer to solve constants or strings byte for byte. While these techniques work well to solve many constraints, more complex ones, such as checksums or encryption, still stall such attempts, as the input's transformation is too complex to follow for such heuristics. In such cases, the fuzzer needs more *domain knowledge*, i. e., information on the specifics of the input format or transformations that are applied to it. Such domain knowledge can be provided in the form of a human expert, specifically tailoring the fuzzer to the system under test, a grammar or specification of the input [10, 151], heavyweight program analysis techniques [33, 96, 143, 201], or approaches harnessing existing domain knowledge [18].

Beyond solving individual constraints to unlock deeper code parts, considering the expected structure of program input is often worthwhile: Programs such as interpreters, databases, or compilers expect a very specific, highly structured input. Many of these targets require the input to follow a certain specification, and the program's parsing stage, usually occurring early during an execution, rejects non-conforming inputs before any deeper program logic can be tested. Traditional byte-oriented mutations of an input, such as bitflips, do not produce another valid input, rendering fuzzing ineffective. Here, grammar-based fuzzing [10, 77, 189] provides a solution where the fuzzer is

provided with a grammar or specification describing how the input should look like. This way, it can generate inputs conforming to the expectations and test the program effectively. If no grammar is available, *grammar inference* approaches, where the fuzzer approximates a grammar on the fly, can help [22, 60, 76].

2.1.3 Heavyweight Feedback-driven Fuzzers

Also called *hybrid fuzzing*, this category of fuzzers pairs the traditional coverage-guided fuzzer with some sophisticated program analysis technique. Two popular examples are symbolic (or concolic) execution [70, 96, 143, 176, 201] and taint tracking [33, 67, 145, 190]. The former is used to sample inputs triggering paths not yet visited by collecting and solving path constraints, while the latter attempts to establish a correspondence between input bytes and their usage in comparisons. Then, the fuzzer can directly modify the relevant bytes, thus helping it advance.

In practice, the anticipated benefits have not fully manifested or have often been offset by the cost of these heavyweight techniques. While symbolic execution can, in theory, craft an input for every path in the program, it fails to scale to real-world programs. Challenges include the high amount of computation resources necessary, the need to model the environment, and the state explosion problem [14]. In practice, many approaches resort to *concolic* execution and focus on a single path.

2.1.4 Input Generation

To effectively explore a system under test, the fuzzer must generate many inputs. We can differentiate between two categories of doing so: *generational* fuzzing and *mutational* fuzzing. The first is often used in the context of grammar fuzzing, while the second usually applies to traditional byte-oriented fuzzing.

Mutation-based fuzzers use a number of *mutators* to alter an input and derive a new input from an existing one. Typical mutators comprise bitflips, simple arithmetic operations, setting interesting values, splicing inputs together, or randomly stacking a selection of all these mutators (so-called *havoc* mutators). These mutations have proven highly effective in changing inputs when targeting programs using a binary-oriented input format, which are prominently used for benchmarking fuzzers [44, 50, 74, 85, 120].

Generational fuzzing often relies on a *specification* or *grammar*, which it uses to generate conforming inputs from scratch. This is used mainly for fuzzing targets where inputs must have a sophisticated, specific structure. Regular bit-oriented mutations may be inadequate to generate another valid input in such scenarios. For example, when trying to fuzz a language interpreter, such as CPython, the input program should have a valid syntax, as otherwise only the parser is tested (which will reject inputs early due to their invalid syntax). Given one input, a valid program, typical fuzzing mutations are unlikely to generate another valid one that is sufficiently different from the first to yield interesting results. Here, grammar-based fuzzing can help to generate more diverse and valid inputs to test a broader set of features. In cases where no grammar is available, *grammar inference* can help to deduce a grammar on the fly [8, 22, 60, 76].

In summary, the fuzzing landscape consists of many diverse approaches aiming to improve specific components or tailor a fuzzer toward specific targets. This diversity, paired with the wide range of application scenarios, naturally led to a high number of papers proposing different fuzzers over the years.

2.2 Exploitation

In the field of computer security, exploitation refers to the process of using some bug within a program to induce unintended and often unwanted behavior. In this work, our focus is on exploitation following *memory corruption* vulnerabilities [179].

Despite its widespread impact in practice, with a whole industry revolving around finding bugs, the formalization of exploitation is scarce. Dullien [51] attempts to formalize concepts that are informally known among exploiters, such as the *weird machine*. Here, we focus on practical aspects.

2.2.1 Memory Corruption

A fundamental root cause of many bugs, *memory corruption*, is haunting computer security despite decades of ever-improving research and fixing bugs. Especially "low-level" programming languages such as C and C++ leave the task of memory management to the programmer, leading inevitably to bugs related to mismanaged memory. While it is easy to propose dropping all programs in these languages, they form the basis of many of our daily used software, including operating systems themselves. Despite their perils,

developers in need of high performance often resort to these languages, even though in recent years safer languages providing similar performance, such as Rust [150], are increasingly gaining favor.

That said, we can differentiate between *spatial* and *temporal* errors that lead to memory corruption [179]. The first category refers to any bug that allows out-of-bounds accesses to memory, thus corrupting it. The famous *buffer overflows* are one example of this class. The second category, *temporal errors*, captures any error that occurs due to memory accesses that must not happen at this point in time; this can be reads of uninitialized data or using freed objects, referred to as use-after-free. Loosely speaking, any memory-related bug not intended by the original program represents a violation of *memory safety*, potentially leading to memory corruption. While such bugs often go unnoticed or simply crash the program, gifted exploiters can turn many of these bugs into a primitive allowing an attacker to do all sorts of unwanted things in the context of the affected program, such as executing arbitrary code.

We differentiate between three fundamental attack classes, code injection, code reuse, and data-only attacks, and a multitude of mitigations proposed and deployed to prevent such attacks. In the following, we briefly introduce them.

2.2.2 Code Injection

The first memory corruption-related bugs, dating back to the nineties [4], were relatively easy to exploit, with little to no protections in place. The most famous attack is the stack-based buffer overflow, where an attacker writes more bytes into a buffer, a local variable living on the stack, than it can hold. The von Neumann computer architecture [188] that has been prevalent in modern computers mixes program code and program data: Throughout the program execution, software uses the stack to store both control data, such as return addresses, and program data, including local variables. Consequently, an attacker capable of writing arbitrary data to the stack exceeding the buffer can not only affect program data but also overwrite control data.

Without mitigations in place, an attacker can simply overwrite the return address of the current function, such that the program's control flow is redirected to a point the attacker can choose once the vulnerable function returns. As the attacker can simply place *shellcode* in the program's memory and jump to it, they can execute arbitrary code, such as spawning a shell. Countermeasures quickly moved to mitigate this easy-

to-exploit vector, and, amongst others, the introduced protection prevents attackers from executing code that lives on the stack [140].

2.2.3 Code Reuse

With the easy way being barred, offensive research then turned its focus toward *reusing* existing code parts. The first famous type of attacks in this category are so-called *return-to-libc* attacks [172], where an attacker redirects the control flow to a function in the standard library. Conveniently, this library is linked into every C/C++ program, respectively, and with functions such as `system()`, it provides valuable primitives to an attacker.

Pushing the idea of returning to existing code even further, attackers can execute arbitrary operations [167] without returning to a function start. Instead, they can scan the program for *gadgets* to chain. A gadget is a short sequence of assembly instructions, usually ending in a `ret` statement. Every gadget fulfills a specific purpose, such as moving a value to another register. Then, the small snippet returns, reading the next attacker-controlled address from the stack, and thus jumps to the next gadget and so on. This chaining of gadgets ending with a `ret` statement is known as *Return-Oriented Programming (ROP)* [148]. Effectively, computing arbitrary operations becomes the task of identifying suitable gadgets. Shacham has shown ROP is Turing-complete given sufficiently large programs or libraries, such as `libc` [167].

Further research has focused on minimizing gadget size [92], proposing alternatives to `ret`, such as jumps (Jump-Oriented Programming, JOP) [25] or C++ virtual functions (Counterfeit Object-Oriented Programming, COOP) [162], or demonstrating ROP does not require knowledge of the attacked program (Blind ROP, BROP) [21] as well as showing that even fine-grained randomization is ineffective [171].

2.2.4 Data-only Attacks

When increasingly stronger mitigations were proposed and employed, attackers analyzed whether program exploitation is feasible without having to hijack the program's control flow. And indeed, research has shown that the capability to overwrite *only* program data is sufficient for exploitation [93, 94].

2.2.5 Mitigations

To detect, stop, or prevent such attacks, many defensive techniques or mitigations have been proposed over the years [95, 179]. In the following, we briefly introduce a subset of techniques that have seen relevant adoption and is interesting for this work.

Data Execution Prevention (DEP) / W^X. This approach aims at preventing code injection attacks by marking memory pages as either *Writable* or *eXecutable* [140, 183], effectively preventing an attacker from executing the shellcode living in memory that the program considers data, such as the stack. This mitigation offers no protection against code reuse attacks, as existing code must be executable by design.

Address Space Layout Randomization (ASLR). The next technique attempts to complicate exploitation by making addresses, such as those of gadgets, unpredictable across different executions. To do so, program segments, both of code and data, are loaded at random offsets upon program start [141]. This method's effectiveness depends on the search space within which an attacker has to find the correct address and the absence of *information leaks*. For example, the search space on 32-bit operating systems can be too small, so attackers can simply use a brute force-style attack to guess the correct address within a reasonable time [170]. However, even when the address space is large enough, any vulnerability leaking information may help the attacker to infer the desired addresses [179].

Stack canaries. Stack canaries [42], akin to canaries used in coal mines, refer to values placed on the stack between user data and control data, such as return addresses. Before returning, the canary value is checked, and, in case of differences, the execution is aborted. This way, an attacker overwriting a buffer on the stack cannot overwrite a return address unless they know the canary value.

Shadow stacks. Shadow stacks [29] are used to store a copy of control-flow data, such as return addresses. Similar to stack canaries, we can then check whether the return address has been tampered with by comparing the one on the stack with the one on the shadow stack. Upon a mismatch, execution is aborted. Thus, simply overwriting a buffer on the stack to change a return address no longer works, requiring an attacker

to resort to different means or modify the return address copied to the shadow stack, too.

Control-Flow Integrity (CFI). CFI [1, 28] aims to ensure that only valid control flow transitions are taken. Often, code-reuse attacks redirect the control flow to another function or piece of code that was never intended to be jumped to from this location. Here, CFI ensures that only valid transitions take place, focusing on forward edges. In other words, upon jumps or calls, CFI validates that the destination may be jumped to from the current location.

2.3 Obfuscation

Software obfuscation, descending from the Latin word *obfuscare* (to darken), aims to impede the analysis of (parts of) code. In general, we refer to any transformation that makes code more complicated and less intelligible as *obfuscation* if it preserves the code's observable input-output behavior. Obfuscation is only one technique in the arsenal of software protection: Collberg et al. [39] propose obfuscation alongside encryption, (partial) server-side execution, and trusted, tamper-proof code as countermeasures. Here, encryption relies on trusted execution environments, for example, Intel's SGX [16], that promise to execute code securely even if the operating system is compromised.

2.3.1 MATE Threat Model

Obfuscation joins the ranks of *software protection techniques* that assume a so-called Man-At-The-End (MATE) attacker model [37]. This scenario models the attacker as a regular end user and, in particular, assumes that they are in possession of the protected binary and can execute, trace, pause, inspect, or modify the software at will. Given these strong capabilities, an attacker usually follows one of the following goals [37]:

1. *Original or simplified code*: An analyst may strive to extract and understand the protected code, for example, to steal intellectual property.

2. *Metadata*: An analyst may want to understand if some code was obfuscated and which techniques or tools have been used in the process.

3. *Data location*: An attacker may desire to extract some data, such as a key embedded in a Digital Rights Management (DRM) system.

4. *Code location*: Similarly, the analyst may want to identify a function executing specific code.

Subsequent work has summarized the first two goals as *code understanding* [16, 159]. As visible from these goals, the underlying motivation is often not uncovering bugs or exploiting an application; usually, the attacker already has full control and respective privileges on their device, making such goals less relevant*. Instead, the focus is on *intellectual property* in the form of code or data in the program. Competitors may desire to steal effective algorithms instead of designing their own or interface with some APIs contrary to the owner's intentions. Similarly, users may wish to crack software, bypass DRM systems to access content without paying, or get around anti-cheat systems to gain an unfair advantage in online gaming.

2.3.2 Obfuscation Techniques

A multitude of obfuscation techniques exist to protect intellectual property in software [16, 36, 159]. In the following, we focus on two techniques that are used in practice, with a particular emphasis on Virtual Machine (VM)-based obfuscation.

VM-based obfuscation. One core technique, VM-based or virtualization-based obfuscation [55, 149], protects parts of a program by translating the to-be-protected code into instructions for a new, custom instruction set architecture (ISA). Usually, this ISA is randomly generated and, thus, only known by the obfuscator. After the code is translated into this new ISA, it is placed as a so-called bytecode in the original program. To facilitate its execution, the program is equipped with an *interpreter* that implements a *fetch-decode-execute* loop for the custom ISA. To do so, it features numerous *handlers*, each implementing one operation. For example, one handle could implement an *addition* of two arguments in such a handler. An attacker facing such obfuscation must first reverse engineer the interpreter of the custom ISA, understand the instruction encoding, and then write a disassembler for this ISA before they can reconstruct the original

* An interesting exception is mobile devices such as iPhones where exploits are required to *jailbreak* the device and obtain root privileges that are otherwise not available to the end user.

high-level code. This highly complex challenge makes the VM-based technique one of the strongest obfuscation schemes. It is, thus, an integral part of many commercial obfuscators, such as Themida [136], VMProtect [187], and others [48, 173].

Mixed Boolean-Arithmetic (MBA). A technique to obfuscate individual expressions are so-called *Mixed Boolean-Arithmetic (MBA) expressions*, which Zhou et al. introduced et al. [208]. The core idea is to transform an expression into a semantically equal yet syntactically more complex one. In other words, the obfuscated expression, referred to as MBA, still computes the same value but consists of more operations that hide the actual computation [54, 208]. To achieve this goal, Zhou et al. define a Boolean-arithmetic algebra comprising arithmetic n-bit operations and bitwise operations, and they define MBA expressions as polynomial functions over this BA $[n]$, to which they then apply transformations (called MBA Transforms). To make an example of an MBA, $(x \oplus y) + 2 \cdot (x \land y)$ is an equivalent but more complex representation of $x + y$.

2.3.3 Automated Deobfuscation Attacks

Many approaches to deobfuscate code exist. We briefly discuss relevant ones:

2.3.3.1 Instruction Removal

A crucial insight is that code obfuscation usually increases the program's complexity (and thus decreases its intelligibility) by adding code that is not relevant to the original one. With this in mind, *instruction removal* attempts to find and remove instructions irrelevant to the actual computation. To this end, there are several strategies:

Compiler-like optimization techniques. Interestingly, simplifying obfuscated code is not too different from optimizing code during compilation: Both aim to find a shorter, simpler version of the code with the same semantics. Consequently, deobfuscation can use well-known techniques such as *dead-code elimination*, *constant folding*, *constant propagation*, or *peephole optimization* [69, 109].

Forward taint analysis. Designed to track user input across an execution, taint analysis [17, 168, 196, 198] can be used to identify code that does *not* contribute to

the result of a VM handler or other unit of code. This way, an attacker can identify statements that have been added with the sole purpose of complicating analysis. To this end, this analysis starts with one or more tainted input variables and then propagates the taint to any value that depends on a tainted value [163].

Backward slicing. Working in the opposite direction of the forward taint analysis, *backward slicing* [43, 192, 198] starts from some output variable and identifies any instruction contributing to the computation of this particular output. The chosen output can, for example, be the output of a VM handler, allowing to strip code unrelated to the handler's actual computation.

2.3.3.2 Symbolic Execution

Various approaches [109, 154, 197] rely on symbolic execution to simplify code. Essentially, such execution abstracts from assembly instructions working on concrete values to a high-level logical behavior of code [163]. For example, when encountering a branch, symbolic execution differs between two categories of inputs based on whether the branch is taken. This way, it can reason on all possible executions rather than a single one. Internally, it uses a state map to track assignments of symbols to variables and memory. It usually works on an intermediate language (IL) that makes side effects explicit, such as modifications of the flag register. To explore the program, it collects branch constraints and uses an SMT solver to solve these, allowing it to generate inputs for both cases. This ability to solve constraints is used in various domains, including fuzzing. In many cases, we use a variant of symbolic execution called *concolic execution* that works on concrete values (rather than a purely symbolic state), helping with precision and potentially simplifying the logical formulas, thus reducing the search space and increasing performance [163]. Traditionally, pure symbolic execution struggles when it cannot infer a formula for code outside the target binary, for example, system calls, which concolic execution can help mitigate by simply executing this call with concrete values [15, 134]. For code deobfuscation, we are often less interested in generating inputs that explore the whole program but want to extract the semantics of obfuscated code. For example, we may want to extract the semantics of a VM handler [109] or follow user input through an execution [154, 197, 198]. For these cases, we rely on the simplification rules many modern symbolic execution engines feature, which allow

them, for example, to propagate constants and make them aware of arithmetic identities. Here, a major challenge is expressions for which the syntactic complexity has been increased, for example, after the application of MBAs.

2.3.3.3 Program Synthesis

Compared to the previous deobfuscation attacks, program synthesis shifts the focus from analyzing the obfuscated code to retrieving the original code *without* having to study the inner workings of the underlying obfuscation itself [23, 45, 119]. Instead, an attacker relies on the fundamental principle that obfuscation may change the syntax of code but not its semantics. In other words, regardless of the protected code's complexity, its observable behavior must remain the same. Based on this insight, an attacker can sample input-output pairs and use *synthesis* to generate an equivalent expression. Many approaches to synthesizing expressions exist [78], including stochastic [80, 102, 158], enumerative [6, 59], deductive [115, 144, 186], or logical ones using constraint-solving [71, 79, 174].

2.3.3.4 Semantic Codebook Checks

Attackers have noticed that VM handlers are often simple in nature [23, 149], with their operation being represented as an *addition* or *subtraction*. Relying on this insight, they have derived a list of basic operations, such as those mentioned, and use an SMT solver to compare the obfuscated code against each of their entries. The SMT solver can then prove that the two pieces of code execute the same operation, allowing the attacker to deduce the code's function without studying the obfuscation [185]. If the code does not match, the attacker proceeds with the next entry in the list. However, this process quickly becomes cumbersome if VM handler semantics are complex and target-specific, as the attacker must compile a list thereof.

Actors interested in automatically simplifying automated code can achieve surprising results using these techniques. Table 1 in Chapter C shows that even simple analysis passes, such as compiler-like optimization techniques, can remove a significant portion of obfuscated code from software protected by commercial state-of-the-art tools like Themida or VMProtect. This drastically simplifies the analysis of handlers, weakening the provided protection.

Reproducibility, Flexibility Requirements, and Risks

Scaling program analysis to practical scenarios requires automation. While many different program analysis techniques and applications exist, we contribute to three different aspects. First, we study fuzzing as a widely popular and successful automated bug-finding technique, analyzing how the presented techniques published at prestigious academic venues are evaluated and whether they are reproducible. Then, we turn towards improving the flexibility of automatically generating exploits. Finally, we study potentially harmful applications of automated techniques using software protection in the form of obfuscation as a case study, and we propose an improved design that is resilient against automated deobfuscation techniques. In the following, we introduce each contribution and briefly summarize it before then discussing its results, limitations, and future work.

3.1 Reproducibility of Fuzzing

Ultimately, automation aims to enable program analysis techniques to scale to large workloads or program sizes a human analyst can not process in a reasonable time. In particular, the capability to automatically find bugs in new code would be a significant step towards improving the security of the overall software ecosystem. Given the amount of software produced daily, automated solutions are required to scale testing successfully and, thus, find bugs. Crucially, automated techniques have the significant advantage that they are cheaper than manual audits by experts, and their results are

usually available faster. The latter property especially allows the inclusion of automated security testing as part of the release cycle. Given a low cost, they could even be part of the *continuous integration (CI)*, testing every single commit. However, before doing so, we must first settle on a suitable testing technique that matches our specific use case. Thus, this process inherently relies on the *reproducibility* of proposed research works. Without such reproducibility, we cannot trust a technique to provide the anticipated benefit, making them unfit for industry adoption.

For the scientific community, the concept of *reproducibility* is equally important: With a constant influx of papers proposing novel techniques or improving existing ones, it is often hard to identify the "best" technique, whether for researchers to test yet another new technique against or for industry practitioners to adopt. For example, when surveying how many papers have been published in the field of fuzzing at prestigious A* [137] security and general software engineering conferences between 2018–2023, we count 289 proposing a new fuzzer or improving at least an aspect of it. The sheer number and the often abstract description make it challenging to easily identify one fuzzer as "the best". Only the ability to reproduce claimed results and compare a technique against others allows others to identify fuzzers that stand the test of time or are suited to particular tasks.

The first contribution of this thesis is a study of how many scientific papers are *reproducible*. To keep the number of analyzed papers manageable, we only include papers published in recent years, focusing on the time between 2018 and 2023. This is because science is continuously evolving, including the implicit, general understanding of how a state-of-the-art research paper should look like, how it should approach a problem, or how its evaluation should be structured. In this sense, some research works do not propose new techniques but reflect on others [19, 130] or a whole field and propose guidelines [2, 9, 38, 47, 104, 117, 184] that help others avoid common errors. Consequently, older papers may contain flaws that were not obvious at the time, while newer papers profit from the accumulated knowledge. Our goal is to study the current state of the art, so we restrict our view to recent papers. We choose 2018 as a lower bound, as it is the year the influential paper outlining how to evaluate fuzz testing by Klees et al. [105] has been published. We also limit the type of papers studied to a single topic, *fuzzing*: The type and scope of evaluations often differ significantly between different fields, making a comparison between them challenging. Fuzzing is well-suited for reproducibility studies, as fuzzing evaluations are highly empirical. The

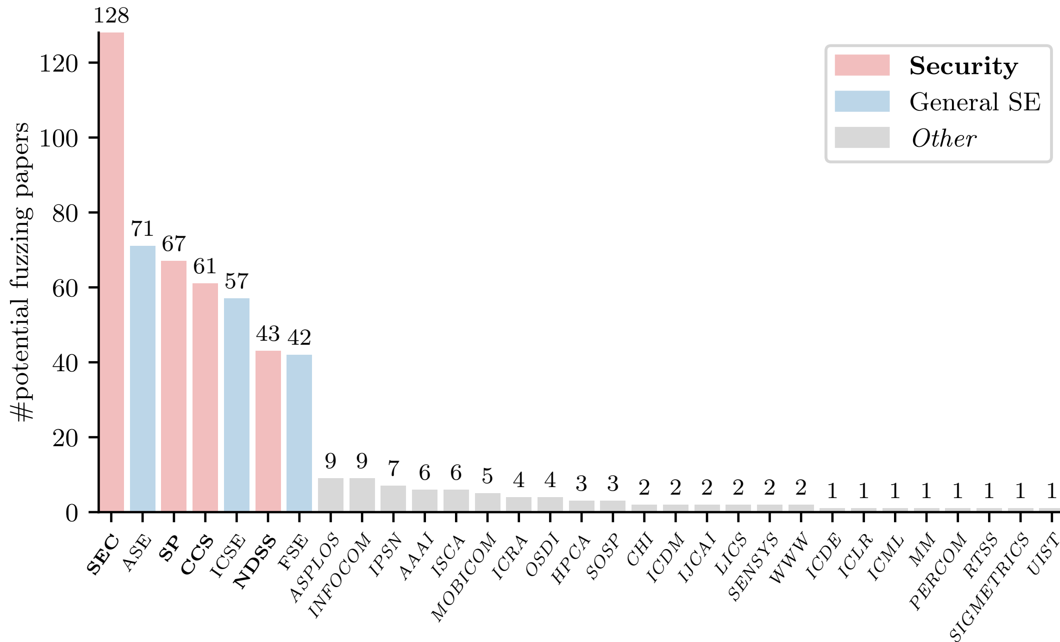


Figure 3.1: Papers on A* venues matching the regular expression `fuzz[^y]` in abstract or title (SEC = USENIX Security Symposium). Note that this only approximates the actual number of papers relevant to our analysis, as papers may use fuzzing for unrelated purposes.

inherent randomness during fuzzing makes it mandatory to extensively test how a fuzzer performs in practice. It also opens the gate for a range of subtle flaws that threaten the evaluation compared to deterministic tools that always (re-)produce the same results. Beyond that, fuzzing is a field that sees many techniques proposed, with almost every technique being presented as a new fuzzer, i. e., we have many tools that can be individually executed and compared against each other. Crucially, fuzzers depend on high performance, meaning that even small implementation details can have a relevant impact.

In summary, fuzzing offers itself to such analysis due to the many tools proposed, their automated fashion, and the potential for subtle errors to threaten the results of an evaluation. Interestingly, the high number of new techniques proposed and the surprising effectiveness in uncovering bugs are contrasted by the low industry adoption. Excluding large tech companies such as Google or Meta, fuzzing has not been broadly adopted to test products.

Studying the *reproducibility* of fuzzing papers, we first conduct a literature survey of 150 out of 289 fuzzing papers published on A* venues (according to the CORE2023

ranking [137]) between 2018 and 2023. Venue-wise, we focus on security and general software engineering venues, which are most likely to contain fuzzing works. To review this decision, we study how many publications on A* venues match `fuzz[^y]`* in their abstract or title (as available on Semantic Scholar [5]). As visible in Figure 3.1, security and general software engineering venues are significantly more likely to publish fuzzing works than other conferences. Note that this only approximates the true number of fuzzing papers, but we believe the magnitudes to be representative. For our literature survey, we have studied how these works conduct their fuzzing evaluations, whether they follow established best practices and guidelines, such as those by Klees et al. [104], FuzzBench [120], or Böhme et al. [30], or if they deviate from them. This way, we can quantify the adoption, identify potential pitfalls and shortcomings of concurrent evaluations, and revise recommendations for future work in the field. Beyond our comprehensive literature survey of roughly half of all fuzzing papers published in the past six years, we select eight papers and attempt to reproduce parts of their experiments in practice. Notably, these works have all caught our attention while reading the respective paper or studying the accompanying artifact, such as when using non-common metrics. This way, we can study if deviating from the best practices can create problems w.r.t. reproducibility.

3.1.1 Key Results

Overall, we find much room for improvement in designing a reproducible evaluation. In general, we find several issues, ranging from a limited choice of baselines to compare against to unfair evaluation setups that distribute resources unequally to the misuse of CVEs as a proxy for impact in practice. The robustness of many evaluations is also dire, with many omitting a statistical evaluation, using too few trials to achieve robust outcomes, or not capturing uncertainty in their empirical measurements. To study whether such problems impact reproducibility in practice, we attempted to reproduce eight papers where we noticed potential pitfalls. We find that even minor errors may threaten the validity and reproducibility of an evaluation, making a careful setup and execution critical. To support future work, we provide updated guidelines and action items authors can follow. For detailed results of this work, we refer to the full publication

* We explicitly avoid matching *"fuzzy"*, which is not used in the context of fuzzing.

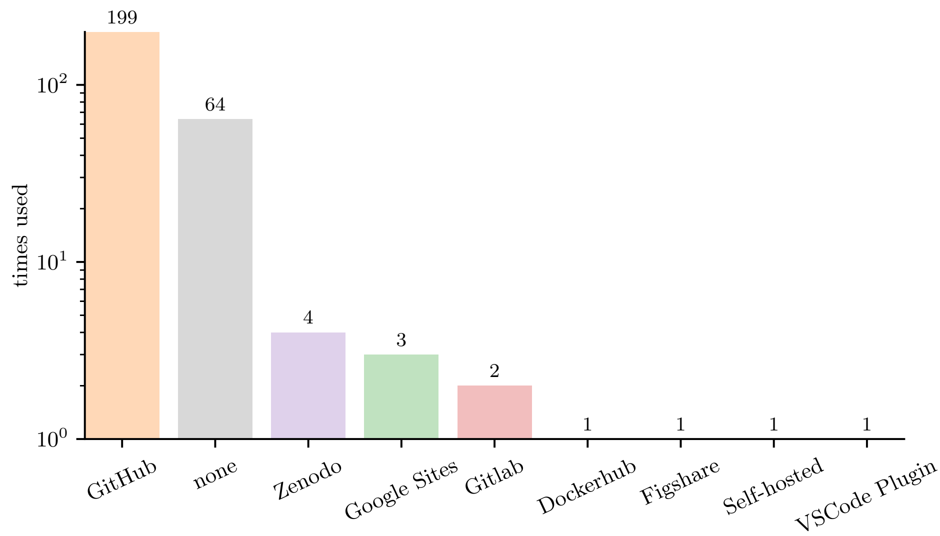
included in Appendix A. In the following, we highlight two surprising key results and extend the discussion of some of our findings.

Source code availability. Reproduction is only possible if the implementation of a new technique is publicly available. An interesting result of our study is that a majority of fuzzing papers already release the source code; however, comparably fewer partake in an artifact evaluation. This is surprising, as the difference between releasing an artifact and participating in the artifact evaluation process, at least when aiming only for the *available* badge, appears small. The only differences are the requirements to use a stable platform for hosting the artifact, to supply some documentation, and the organizational overhead of applying for the badge.

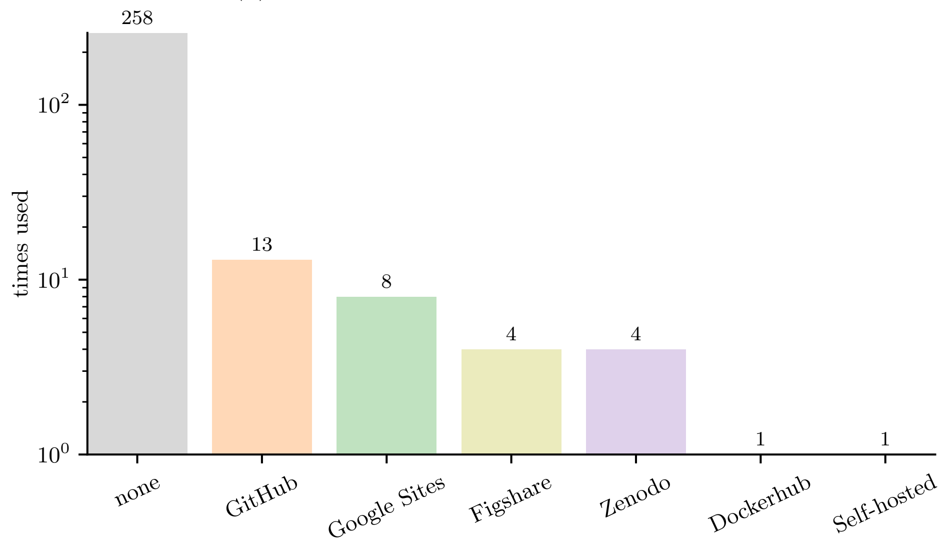
To identify whether the requirement of publishing stable code is at fault, we extend upon the results presented in Appendix A and study the used hosting providers across analyzed papers. As the results in Figure 3.2 show, we find GitHub to be the most prominent choice, used by 94% of papers that release their code and 42% of papers that release data. As GitHub is already considered a stable hosting provider, it is unlikely this is the reason for the observed discrepancy between released and formally *available* artifacts. Anecdotally, we have seen some code repositories with little or no documentation or cases where only a subset of code is made available; however, the large majority have been shipped with at least a rudimentary README.md file that provides basic documentation. Based on personal experience, the organizational overhead of applying for a badge has also been acceptable, raising the question as to why artifact evaluation is not more widespread. We note that the whole process is relatively new and not yet offered by all conferences. Conferences in the security field have only introduced artifact evaluation in recent years, with CCS doing so as recently as 2023. Consequently, authors may simply lack familiarity and awareness of this process, making increased participation more likely as years pass by.

Common Vulnerabilities and Exposures (CVEs). Another key result has been the (mis-)use of CVEs as a proxy for real-world impact. In the following, we extend our analysis and discussion.

As a brief background, in 1999, Mitre initiated a program to standardize the enumeration of vulnerabilities [114], which quickly found widespread industry adoption. Essentially, they hand out CVEs, unique identifiers for vulnerabilities that help vendors



(a) Platforms used for publishing code



(b) Platforms used for publishing data

Figure 3.2: Logarithmic plots showing the platforms that 289 fuzzing papers use to publish their source code or accompanying data (if any). The longevity of research artifacts is a concern for reproducibility.

and users efficiently communicate about a specific bug or discover whether a particular product suffers from a vulnerability [123]. Anyone can report an identified flaw in some software. Then, the CVE Numbering Authority (CNA) responsible for the affected software, such as Microsoft for Windows or Google for Chrome, validates the report and assigns a CVE that allows everybody to recognize and track this particular flaw. If no CNA exists for some product, e. g., some open-source software, Mitre itself reasons on whether to assign a CVE as a last-ditch effort. This is usually only necessary for open-source software or smaller vendors, as large tech companies have signed up as the respective CNAs for their products. Crucially, CVEs are designed as *identifiers* of vulnerabilities, not as proof of vulnerability, measurement of impact, or reward for reporting a bug. Orthogonal to CVEs, the Common Vulnerability Scoring System (CVSS) [64] standard can be used to assign an impact score, as NIST does in the National Vulnerability Database (NVD) [129].

Now, as our work in Chapter A thematizes, authors of fuzzing papers use CVEs to show the practical impact of their new tool. In particular, 59 of the 150 papers claimed at least one CVE, with a total of 662 new CVEs being reported across them. Anecdotally, reviewers of security conferences have also asked us for CVEs during paper submission. All this suggests that CVEs since have become a metric to measure if some tool can find bugs in real-world software. Unfortunately, the discrepancy between original intention and current use opens the gate for abuse. Authors of academic work now have an incentive to try and obtain as many CVEs as possible, purportedly boosting the perceived impact of their work. At the same time, the CVE assignment process does not necessarily include a verification of the soundness and reasonableness of a bug. While companies acting as CNA for their product have a clear incentive to verify bugs prior to assigning CVEs[†], this is not the case if no CNA (other than Mitre in its function as fallback) exists. Mitre faces a dilemma for such reports: Either they attempt to verify the validity of reports for bugs in foreign software, potentially opting not to assign a CVE for a vulnerability they might not fully understand, or they conservatively assign a CVE to any plausible report, thereby assigning some CVEs even if no underlying vulnerability exists. Based on personal experience (and as subsequent experiments show), they understandably chose to err on the conservative side. However, this combination of an incentive to obtain many CVEs paired with a lack

[†] In fact, they may even opt not to assign a CVE despite the reported bug being a vulnerability that per se would deserve one. We do not study this type of misuse in this work.

of verification is exactly what leads to a potentially subpar quality of assigned CVEs reported in papers, as actual security considerations take a backseat.

Investigating the 339 claimed by 35 papers, we found less than half of them to be *valid* and *fixed* (or *acknowledged*). We consider a CVE valid if its formal status is *published*, as opposed to *disputed* or *rejected*, and we consider it *fixed* (or *acknowledged*) once the developers fix (or at least confirm) the presence and veracity of the reported vulnerability. As visible, the CVE system provides means to challenge a published CVE: Maintainers can *dispute* a CVE, meaning they doubt its veracity but Mitre sees grounds for discussion. If Mitre sees no such ground, a CVE is officially *rejected*.

When looking at the 57% of CVEs that are not valid, we find they are either *reserved* (i. e., still blinded as the underlying bug has not been fixed nor disclosed) or they have been refuted, disputed, or simply ignored by the maintainers of the underlying software. More precisely, the maintainers ignored 20% of all CVEs (69) and considered 11% (37) as invalid, for example, because the reported bug was not security relevant. Focusing on the ignored CVEs, we find that 14 of the 69 CVEs have been assigned for no longer maintained projects. While some of these projects have been superseded, such as Contiki by Contiki-NG, others are small GitHub projects that are seemingly hobby projects not used in production scenarios. While testing software for bugs is generally favorable, the benefit of testing unknown, unused, and deprecated projects remains marginal. It raises the question of whether the goal was to make the particular software more secure or to increase the perceived practical impact by increasing the number of assigned CVEs. That said, it may be difficult to assess who still uses or relies on a particular software, even if it no longer receives updates.

However, even when the software maintainers do not ignore a report, we empirically found that they are often conservative when it comes to fuzzer-generated bug reports. Some doubt that a bug would occur in practice, assuming that a regular user would never supply similar data. In general, this assumption is flawed as a malicious user may not behave like a regular user. One example of such behavior is cases where a user can set the packet size of transferred packets. Suppose a server does not validate the size but trusts the user's specification. Then, an attacker can easily provoke out-of-bounds memory accesses by setting a packet size deviating from the real one. In other cases, maintainer doubts may be justified: For example, finding memory leaks in short-running client-side programs that are never exposed to the outside bears little risk per se. Researchers trying to obtain a CVE for such a bug may be perceived as

pushing the number of credited CVEs rather than actually caring about securing the software or its users. It is generally difficult to estimate whether a given bug report has real-world impact or not: Reporters may lack a deeper understanding of the analyzed code, while maintainers or developers may lack awareness regarding security matters. Other developers are only discontent with the style of reporting: A fuzzer often finds many crashing inputs, particularly if the software has been fuzzed for the first time. If the findings are not manually deduplicated but instead dumped in the project's bug tracker, this results in a large number of "low effort" reports that make little to no attempt to identify the underlying root cause or propose a fix. With unclear impact, no apparent use case (as fuzzing input often appears random), and little additional information, maintainers are often more annoyed than happy to receive such reports, with one calling this type of CVEs even "fuzzer fake CVEs" [118]. In these cases, it appears high likely that the CVE has been requested by the authors finding the bug rather than the maintainers themselves, underscoring the existence of this hunt for CVEs. When contrasting the number of CVEs that are invalid (36) with the number of CVEs that have been officially disputed (1) or rejected (2), we find that maintainers are unlikely to formally contest bad CVEs but instead simply close the respective bug report. Consequently, 73% of assigned CVEs are still considered valid, even though only 43% have been acknowledged or fixed by the maintainers.

Now, this raises many questions, most importantly whether academia needs to adjust its use of CVEs. We believe that CVEs are not a suitable metric to show real-world impact, mainly because the assignment process is not connected to a verification of the bug's validity. While companies acting as CNAs for their product are likely to verify bugs before assigning a CVE, Mitre is assigning CVEs without in-depth verification. We verify this notion by studying the CNA assigning the CVE *if* the CVE was ignored or invalid and display the results in Table 3.1: 104 of the 107 bad CVEs have been assigned by Mitre, the remaining three by Redhat. We stress again that this is by design: Mitre functions as a *catch-all* in case no other CNA is responsible. The reported bugs are thus not in products they own but can be in any code, meaning they cannot reasonably verify whether a bug is indeed a bug, nor can they necessarily coordinate with the maintainers. Likewise, Redhat assigns CVEs for various open-source software, leaving them in a similar spot.

With such a discrepancy between assigning authority and maintainers, the system can be gamed by reporting all sorts of bugs in all sorts of software instead of work-

CNA	Outcome	#CVEs	% of CNA
Mitre	Acknowledged	82	44%
	Ignored	69	37%
	Invalid	34	18%
Redhat	Acknowledged	11	79%
	Ignored	0	0%
	Invalid	3	21%
Others*	Acknowledged	52	100%
	Ignored	0	0%
	Invalid	0	0%

* Adobe, Apple, dwf, Github, Google (Android, Chrome), IBM, Microsoft, Oracle, Qualcomm

Table 3.1: Outcome of CVEs assigned by different CVE Numbering Authorities (CNAs). The percentage is relative to all CVEs assigned by this CNA.

ing with the maintainers towards responsible disclosure—leading to CVEs for non-exploitable bugs or in software nobody uses. Currently, about every fifth CVE assigned by Mitre has been refuted by the project maintainers. This serves as a call to action for academia to reconsider the current use of CVEs as a metric or identify suitable mechanisms to avoid blatant misuse.

3.1.2 Discussion

The results of our paper warrant more discussion in many directions. In the following, we focus on the reasons leading to a lack of reproducibility and the future of artifact availability.

Underlying reasons. One fundamental question our published work has not addressed is why evaluations cannot be reproduced: The paper merely *observes* problems regarding reproducibility among scientific papers in the area of fuzzing, and it outlines how future work could compensate for current pitfalls. Here, we attempt to elucidate the *reasons* leading to this problem in the first place:

- *Publish or perish:* The prevailing need to publish as many papers as possible provides little incentive to critically reflect on one's work, instead making authors haste to the next deadline. Without careful review of results, it is easy to miss errors that may falsify evaluation results.

- *Complexity:* Fuzzing evaluations are *hard* to get right. Simple things such as the CPU clock can massively distort outcomes without this bias being trivially detectable or adequately discussed in the literature. This is even true in cases where evaluation platforms such as FuzzBench are used locally, potentially generating a false sense of security.

- *Lack of enforcement:* At the same time, reviews seem to pay little attention to proper documentation of the evaluation setup, creating no incentive for the authors to do so in-depth. Also, reviewers often have no access to the artifact, making judging the used experiment setup or reproducing experiments hard.

In the current state, neither sufficient incentives to reflect on and document one's exact evaluation setup nor repercussions for not doing so exist, making reproducibility an afterthought for many submissions.

Artifact availability. Despite many papers publishing source code, a non-negligible number still does not. We see little reason why academic work should be withheld from publication: In edge cases where releasing an implementation may pose a threat to society or cause unnecessary harm, releasing code may be rightfully forfeit. However, this appears not to apply to most works. On the other hand, not releasing an artifact always reduces the value of a work to science: Neither can others reproduce the results for confirmation or correction, nor can they easily build upon a proposed technique or compare against it. Reimplementing a technique solely based on information a paper provides is costly and may need to fill the gaps in terms of unspecified implementation details, making it unreliable for comparisons. As the academic benefit of a reimplementation is low, it is unattractive, and the paper without source code remains a blocker of follow-up research. Thus, authors should always strive to release artifacts where morally acceptable or, otherwise, document reasons for not doing so.

3.1.3 Limitations

Our approach has two particularly noteworthy limitations, namely its focus on the mechanical correctness of the experiment design and the potential non-applicability of guidelines to edge cases.

Semantic correctness. These guidelines enable *reproducibility* by ensuring the evaluation design is reasonable. However, this is limited to a mechanical level: The guidelines do not allow judging a specific technique's inherent value. Even mechanically reasonable evaluated techniques may not make sense: Imagine a new method aiming at fuzzing library APIs. It is inherently hard to understand the API contracts, i. e., inherent assumptions of how the input to a specific input should look like. When picking a random function and replacing argument values with random fuzzing input, we may trivially find many "bugs". However, without knowing whether this particular function can be reached with such input during regular operations, these findings may be invalid. Other functions may sanitize the input or the arguments may not be user-controllable, potentially rendering all presumed findings invalid. Still, evaluating such an approach in a mechanically correct manner is possible, such that it may appear well-evaluated and successful. Only when considering the actual context does it become apparent that such a technique contributes little to securing real-world software.

Guideline applicability. No matter how well-spirited, any guidelines may fall short of capturing all nuances and edge cases. Evaluating on bugs only makes sense when using fuzzing to find bugs. Future work may derive other use cases, such as using fuzzing for its capability of generating a diverse set of inputs, aiming to enrich other analyses [97, 138]. Similarly, evaluating on a specific metric may not be possible in some scenarios: When fuzzing firmware on a proprietary device in a blind manner [156], we may have no possibility to extract exercised coverage. Crucially, this does not necessarily imply that the work is inherently bad or should be rejected until it measures coverage. In short, any guidelines proposed may apply to a majority of cases, yet, judging a work based on a superficial comparison to guidelines is inherently flawed. Our guidelines propose a helpful set of things to *consider* and, if necessary, ignore based on a case-by-case basis.

3.1.4 Future work

There are multiple avenues future work could take to improve or extend upon this work.

Extended literature analysis. Extending the analysis to all 289 (as opposed to the current selection of 150) may further solidify the results. Beyond A* conferences, multiple other venues, such as the International Symposium on Software Testing and Analysis (ISSTA), feature fuzzing works. While we do not believe that analyzing more works would significantly change the results, differences between venues or particular evaluation flaws may exist that only show in a select number of papers not yet analyzed. In an even broader scope, a reproducibility analysis could be extended to virtually any field where new scientific publications are usually accompanied by some tool that is released. At the same time, future work could email authors who have not publicly released an artifact to understand the reasons for keeping source code private.

Replication. Instead of reproducing eight case studies as we did, all fuzzing papers could undergo external validation. Even better than reproducing the results using the original implementation, *replicating* the results, i. e., implementing the proposed technique anew and independently from the original one [12], could further help identify implementation errors leading to skewed results. The community has already taken this upon themselves, with the AFL++ project [90] implementing select techniques that are promising. Similarly, LibAFL [62] provides a platform on which different techniques can be pieced together. Optimally, significantly more techniques are not only proposed as a separate fuzzer but re-implemented by other community members on top of existing fuzzers, thereby allowing improvements to be clearly attributed to a particular technique rather than implementation-specifics.

3.2 Flexibility for Automated Exploit Generation

Fuzzing is only the first step in the pipeline of automatically finding and addressing bugs: Its findings are usually crashing inputs, functioning as proof that some bug exists within a program.

To properly address a crash, an analyst or developer must determine the underlying bug and, potentially, its root cause. Given a sufficient understanding, they can then derive a proper patch. Now, our goal is to automate as much of this process as possible: Finding a bug, understanding its implications and root cause, and, ideally, automatically providing a fix. Unfortunately, automated program repair is not yet capable of deriving patches of developer quality [20, 116, 207] and often lacks the semantic insight a programmer possesses. When we are unable to resolve the problem automatically, the next best thing we can do is provide the developers with as much helpful information as possible. In particular, we can help them prioritize bugs with critical security impact over unimportant annoyances. One approach to show whether a particular bug is security-sensitive is to take an attacker's point of view and attempt to generate an exploit automatically.

In this vein, we propose a new technique for automated exploit generation that relies on a logic encoding of gadgets that it then chains together using an SMT solver. This contrasts typical approaches that rely on some heuristic that helps them identify "good" gadgets to chain. Our approach demonstrates unprecedented flexibility, as arbitrary constraints can be modeled with little effort. Any constraint that can be expressed as SMT formulas can simply be added, offloading the task of finding a suitable solution to the SMT solver. Another benefit other techniques lack is the SMT solver's capability to prove that a gadget chain cannot exist for the given gadgets and conditions. Where other tools fail, it simply indicates a failure of the provided heuristics, leaving a human analyst wondering whether manual effort would prevail. With our approach giving a mathematical proof, the analyst can focus their effort on other tasks rather than wasting their time.

On a technical level, we propose to encode the data flow between potential gadgets as a logic formula. When combining this formula with an encoding of the CPU and memory state at the time at which control can be transferred to the gadget chains and the desired CPU and memory state that the gadget chain should achieve, we transfer the program of finding a gadget chain into a reachability problem that we

can pass to an SMT solver. If the solver finds a satisfiable assignment for all variables in the formula, it provides a model containing concrete assignments. In other words, the solver managed to find a gadget chain leading to the desired outcome. Due to the logic formula's nature, it is easy to impose arbitrary restrictions on the gadget chain process, such as excluding specific bytes from occurring (so-called *bad bytes*, for example, the null byte which cannot be transferred into the memory by the exploit when relying on functions that accept this byte as string terminator). Other scenarios where our approach's flexibility might come in handy include scenarios where an anti-cheat engine checksums values on the stack. We can simply add constraints that the gadget chain's checksum must match this particular value.

3.2.1 Key Results

The results show that using an SMT solver to find gadget chains is effective. Compared to traditional heuristics, its selected gadgets contain more instructions and show a higher diversity in terms of memory accesses and control flow types used to chain gadgets together. To evaluate its effectiveness, we chose five realistic targets, ranging from `libc` over `chromium` to `apache2`, and attempted to find a chain for three attacker goals. Overall, our prototype, `SGC`, proved more successful than other evaluated tools, and it found a gadget chain in all but one case. In this scenario, it *proved* that no gadget chain exists for the given pre- and postconditions, which no other evaluated tool could do. Further, `SGC`'s gadget chains were more diverse, using different types of control flows and even including gadgets that write or read memory, which heuristics-based tools avoid. Our results showed that the comparably long runtime of the SMT solver can be addressed by randomly sampling a subset of gadgets, which may be required for a high amount of available gadgets. Crucially, any expression that can be encoded as a logical constraint can be taken into account, such as requiring all the addresses of gadgets in the attacker-controlled buffer to add up to a specific checksum. This capability may be interesting in certain scenarios where tools such as DRM systems or anti-cheat engines attempt to enforce the integrity of memory, allowing `SGC` to handle a broader range of scenarios before requiring human intervention.

3.2.2 Discussion

We would like to discuss two key points.

Bug assessment. While our technique provides an approach to automatically generate exploits, it is easy to see how it can be helpful to software developers: When having multiple bug reports that require time to address, identifying such bugs leading to vulnerabilities may be desirable. Developers could use `SGC` to model the capabilities the bug provides them and see whether a gadget chain can be found, signaling exploitability, or whether `SGC` proves that no gadget chain may exist for the given set of pre- and postconditions. While this is no guarantee in terms of non-exploitability, as an attacker may find different pre- or postconditions that yield a valid gadget chain, it may indicate that other bugs should be fixed first. Existing work in that domain uses a fuzzer to identify a set of capabilities an attacker gains from a crashing input [97] or attempts to infer the severity based on available descriptions of vulnerabilities [53, 83]. Heuristics based on already available descriptions provide only an approximation based on existing knowledge, thus primarily helping to identify N-days or vulnerabilities that are similar to known ones. Such approaches are orthogonal to exploring the concrete bug observed, helping to relate it to potentially similar ones. Using a fuzzer is a promising avenue to explore the concrete bug, as any identified capability is demonstrably one that an attacker can use, but is fundamentally more geared towards *understanding the bug* than assessing it. Arguably, the boundaries between understanding and assessing a bug are blurred, as any understanding may contribute towards a more precise assessment. That said, the exploration of the bug provides fundamental capabilities without trying to see whether they can be pieced together in the context of the program, as does automated exploit generation.

Ethical considerations. As with any automated exploit generation approach, the potentially offensive nature of `SGC` raises questions regarding ethics. After all, it is not far-fetched to assume that malicious actors might want to misuse this (and similar) research to build exploits that can be used in practice and harm innocent users. Similar to Heelan [86], we believe this to be an understandable but fundamentally incorrect assumption: Bad actors have been, are currently, and will in the future be actively working on exploitation regardless of whether academic work refrains from publishing on the

matter. Neither our approach nor others provide a full end-to-end technique that automatically finds a bug, identifies the root cause and exploitability, and builds an exploit. Such an end-to-end approach would eliminate any required expert domain knowledge, thus opening exploitation to a wider audience, potentially causing an augmented level of misuse. To our knowledge, no such approach is currently publicly available. Instead, tools such as `SGC` require the domain knowledge of an expert (for example, to identify suitable pre- and postconditions), who are already capable of finding gadget chains manually. Consequently, the choice is between researching the topic and potentially providing malicious but already capable individuals with ideas for their own tools or not researching the topic and remaining in the dark regarding state-of-the-art automated exploit generation. Without research, such works also will not be available to developers intending to use them for assessing bugs, as they have few incentives and resources to develop such tools on their own (compared to malicious actors, which may have enough interest and resources to develop automated exploit generation tools).

3.2.3 Limitations

Our current approach has two noteworthy limitations.

PC control. Currently, our approach—as is common [164]—requires control over the Program Counter to transfer the control initially to the gadget chain. Given some crashing input found by a fuzzer, it is unlikely to lead to such control directly, requiring a human analyst to study the bug, its implications, and potential attacker capabilities; only then can they use our approach to automatically generate an exploit. This gap between initially identifying a bug and working out concrete preconditions leaves ample opportunity for future work to address using automated techniques.

Sampling. When the SMT solver finds no chain, the reported `UNSAT` signals that no chain exists, functioning as proof of non-existence that helps developers judge whether a particular bug should be prioritized. However, the performance characteristics of our approach require us to randomly sample a subset of all available gadgets to speed up the process of deriving a chain. In this case, the `UNSAT` outcome only indicates that no chain exists for the currently sampled gadget pool. This gain in performance leads to a significantly weaker statement that provides little help to assessing bugs and their

criticality. Consequently, developers interested in using our approach cannot rely on sampling as a performance optimization, leading to long runtimes of our approach.

3.2.4 Future Work

Beyond the room for improvement outlined above, we consider the following avenues to worthy of further reserach.

Performance improvements. First, implementation-wise, a more efficient disassembly procedure would significantly improve performance. Our experiments testing three scenarios showed that disassembly consistently averaged around 30 minutes (cf. Table 4 in Chapter B, whereas the solving time averaged to approximately 20 minutes. Notably, solving time depends significantly on the complexity of the attacker goals; for simpler scenarios, it took only 6 and 8 minutes but 45 for the more complex `mmap` chain. In any case, the constant disassembly time of 30 minutes contrasts modern disassembly tools such as IDA Pro, Ghidra, or Binary Ninja, which are usually significantly faster. Similar to improving the implementation in terms of disassembly, it could be worthwhile to study the differences between multiple SMT solvers. Other works have observed significant differences in solving time when switching to another SMT solver [65]. In our case, we use Boolector [133] and rely on its `const-array` extension [177] to model memory, which other solvers may not support. However, Boolector's successor, Bitwutzla [132], could be an interesting option to improve performance.

Mitigation awareness. Second, a highly interesting avenue of future work could focus on extending our approach to model other mitigations and defense mechanisms. In particular, CFI [1] could be amenable to our technique. CFI protects against attack redirecting the control flow by checking if the target of a control flow transition is *valid*. This severely limits the gadget pool and the order in which gadgets can be chained, as any successor must be valid w.r.t. to the CFI policy. The flexibility of our approach should allow its application to this restricted scenario, where gadgets have only a limited set of valid successors. As it stands, our current implementation disassembles gadgets without respect to legal control flow transitions and would require adaption to model valid units of code under a CFI policy.

3.3 Risks of Automation: Weakening Obfuscation

Up to this point, the role of automation has been one of enabling security analysis to scale. However, its use is not restricted to the purpose of finding bugs nor to companies or security-oriented individuals with benign goals in mind. In this section, we highlight one subject where automation has shifted the balance in the arms race between attacker and defender towards the former: Software protection in the form of code obfuscation. Here, the capability to scale deobfuscation through automation poses a major threat to intellectual property. Obfuscation's inherent reliance on security by obscurity makes it challenging to quantify the damage or propose concrete security guarantees. However, if automated analyses succeed in significantly simplifying obfuscated code, it is safe to consider the used obfuscation scheme as unreliable and broken. While all obfuscation can eventually be broken by a human analyst spending enough time on a target, this does not scale to hundreds of different obfuscated binaries and is generally costly. On the other hand, running a simple, automated analysis requires little expert knowledge and can easily be scaled to hundreds of obfuscated binaries. From a defender's point of view, the increasingly more powerful automation of program analyses helping us to scale the finding of bugs is hurting us here.

In our work, we first demonstrate that even comparably simple automated analyses are surprisingly effective in simplifying commercial state of the art: Already a simple dead code analysis removes half of a handler's instructions for commercial obfuscators (see Table 1 in Chapter C), paving the way for subsequent analysis. That said, more complex approaches to deobfuscate these commercial obfuscators exist both in academia [99, 106, 194] and practice [56, 153].

With this state of affairs in mind, our work studies whether increasingly powerful automated analyses can be kept within bounds such that we improve our bug-finding capabilities without sacrificing software protection. To this end, we propose an obfuscation scheme that combines multiple techniques to provide strong protection. This combination achieves *resilience*, i. e., protection against automated analyses, by relying on inherent weaknesses of the program analyses. More precisely, we formulate our goal as preventing an attacker from extracting the *core semantics* from some function f. The core semantics refer to f's functionality, such as an addition of two variables. While abstract and generic in nature, f can be used to model a VM handler, in which case its core semantics would represent the operation implemented by the handler. Our

prototype, in fact, uses a VM-based obfuscation approach to implement our proposed techniques. However, we point out that hiding the VM or making its structure more resilient is orthogonal to our goal: We focus on protecting the VM handlers, or generically, the function f. In particular, we propose three principles that, in combination, promise resilience:

1. Merging core semantics
2. Adding syntactic complexity
3. Adding semantic complexity

Merging core semantics. In the first step, we merge multiple core semantics and intertwine them such that individual semantics can still be invoked using a key, but at the same time, all semantics are always executed. This preserves functionality (as we can still invoke each core semantics individually) but ensures the output always depends on all statements (as each invocation always executes every single statement) such that deobfuscation attacks based on instruction removal can no longer discard statements pertaining to core semantics not invoked for a particular execution. At the same time, we use partial point functions to make the key selection hard to identify and propose to use mathematically hard problems, here factorization, to prevent a static attacker relying on SMT solvers from identifying the invoked core semantics.

Adding syntactic complexity. To make the protected code harder to analyze, we further increase its *syntactic complexity* using MBAs. In other words, we inflate the expression while preserving the original semantics. To this end, we propose a generic approach to generate MBAs of arbitrary depth by recursively rewriting expressions with syntactically more complex ones.

Adding semantic complexity. Program synthesis simply bypasses all syntactic complexity, as it works with input-output samples rather than studying the expression itself. Thus, we need to increase the semantic complexity of our function f to harden it against this attack vector. We propose chaining multiple, program-specific core semantics into a superoperator. For example, suppose a program executes the operations A, B, and C in order. Then, we propose creating a superoperator f_{ABC} executing all three instead of invoking three core semantics f_A, f_B, and f_C in sequence. This causes

our core semantics to be more complex, making it harder for program synthesis to synthesize.

When combining these techniques, synergy effects arise. For example, when adding MBAs to increase the syntactic complexity, their diverse nature also makes it more complicated to identify and pattern-match structures. That said, a risk of any obfuscation scheme is that the ample code transformations falsify the original code's behavior. While formal verification of individual expressions (or short sequences thereof) is possible, a full end-to-end verification of complex code is currently out of scope. Thus, we use formal techniques to verify the correctness of our MBAs and resort to black-box differential fuzzing to test the IL and binary produced by our obfuscator, Loki, comparing it to the original code's behavior.

3.3.1 Key Results

Our results show that it is possible to provide adequate resilience against a wide range of automated analyses, including taint analysis, symbolic execution, and program synthesis, by relying on such a combination of techniques. More precisely, these deobfuscation techniques can simplify significantly fewer expressions than for state of the art. For program synthesis, we show the limits of synthesis, deriving how complex expressions must be to resist such attacks effectively. Throughout our verifying efforts, in particular, our differential fuzzing, we find that none of our transformations changed the semantics of the underlying code. This comes at the cost of higher runtime overhead and a larger memory footprint; still, our work remains in line with state-of-the-art commercial protection schemes. In any case, it is crucial to selectively apply obfuscation to critical pieces of intellectual property and avoid hot paths in software.

As this chapter demonstrates, automation can be used not only to find bugs and improve security but also in the opposite direction. At the same time, our results show that automated analyses are no panacea: Software can be designed explicitly to thwart automated analysis. In the case of software protection, this may be desirable; however, developers may similarly design software to prevent automated analysis of their code, similar to the techniques proposed by AntiFuzz [81] or Fuzzification [98].

3.3.2 Discussion

The nature of the arms race between deobfuscation and obfuscation as well as the relationship of software protection to bug finding warrant further discussion.

Longevity. While our combination of techniques focuses explicitly on exploiting inherent weaknesses of analyses, fast-moving advancements in computing power or new analyses may threaten the security guarantees provided. The current situation resembles an arms race where more powerful techniques are quickly followed suit by stronger deobfuscation techniques. Given the dynamic nature, a natural concern is whether Loki will provide lasting resilience, assuming newer and stronger deobfuscation techniques are already worked on, for example, in terms of MBA deobfuscation [108, 146, 147]: Here, new approaches grounded in math show promising results in simplifying MBAs applied to simple expressions such as $x + y$. Ultimately, predicting the longevity of Loki, or obfuscation in general, is difficult. In any case, faced with more potent attacks, obfuscators, including Loki, have to ramp up the complexity of obfuscated code. For example, Loki allows to set parameters such as the rewriting bound or depth of core semantics. Additionally, obfuscators can incorporate new obfuscation techniques focusing on weaknesses of new analysis techniques. At the same time, more complexity further hurts the performance, making it paramount to study orthogonal protection techniques that can extend obfuscation.

Bug pipeline. Now, the astute reader may wonder how obfuscation relates to the bug pipeline or fuzzing in the first place, if at all. Intuitively, there is a direct connection: The former attempts to make the analysis of programs more difficult, while the latter analyzes programs to find bugs. On the contrary, their usage scenarios widely differ: Fuzzing is often used by companies or researchers intending to scan as many programs as possible, while obfuscation has the goal of protecting small parts of code from prying eyes. Crucially, the code's interestingness often stems not from potential bugs but its inner workings, with the attacker's goal being to steal intellectual property or bypass some anti-cheat or DRM mechanism. Using a fuzzer barely helps an attacker come closer to their goal in such cases, so obfuscation has little interest in preventing fuzzing in the first place. Notably, this does not hold true if obfuscation is chosen intentionally to make code harder to read for fear of an analyst finding bugs. While not obfuscation's

primary application, some companies may choose this path to protect code with a large attack surface.

That said, many obfuscation techniques are inherently unsuited to impede automated bug finding in the form of fuzzing: Anything that only makes code more complex to read has little impact on the fuzzer other than decreasing the number of executions due to degraded performance. More powerful techniques, such as VM-based obfuscation, may superficially appear suited to prevent fuzzing. After all, the fuzzer's coverage feedback is limited to the VM handlers, as it has no concept of the virtualized code and, thus, may not fully explore it. This is similar to the case of fuzzing interpreted languages, e. g., Python, where the fuzzer's feedback is limited to the interpreter and receives little feedback on the interpreted code itself. However, even simply identifying the virtual instruction pointer and using it as additional feedback to the fuzzer is sufficient to overcome this [193]. Güler et al. have further evaluated how fuzzers can find bugs in the presence of various obfuscation schemes, with their findings being in line with our expectations [81].

Unsurprisingly, concerning automated exploit generation, obfuscation has been shown to have negative consequences, as its increase in code size, i. e., the added instructions, leads to a larger gadget pool [206]. Given a larger, more diverse gadget pool, automated exploit generation tools are more likely to find a gadget chain.

Diversity. Beyond offering solid protection against automated and manual attacks while imposing as little overhead as possible, there is another dimension to obfuscation: diversity. When an attacker succeeds in deobfuscating one binary, we do not want this to translate to all binaries protected by the same technique being broken. Instead, deobufscating each binary should always incur a significant cost, thereby ensuring that other obfuscated binaries are protected even if the obfuscation is broken once. To ensure diversity of instances, Loki relies on randomness at various points: Both superoperators and MBAs are sampled from a large search space, such that two instances differ from each other. Crucially, we avoid hardcoded rules that limit diversity. That said, a high diversity such as that provided by Loki is critical to thwart pattern matching, preventing attackers from simply matching for identifiable snippets of code.

3.3.3 Limitations

Two core limitations of our approach are the restriction of our focus on resilience rather than robustness or secrecy [39] and the shortcomings of our prototype.

Potency and stealth. Our work has put a strong focus on *resilience*, i. e., the capability of thwarting *automated* deobfuscation attacks [39], rather than *potency*, its capability to confuse human analysts. Intuitively, there is a relationship between the two: An attacker that can use automated attacks will succeed faster at deobfuscating protected code than one that has to first understand the code obfuscation techniques used. However, common wisdom among practitioners is that a human attacker with enough time will always succeed in finding and overcoming code obfuscation [149]. That said, measuring potency is difficult, with the capabilities of a human attacker strongly depending on their knowledge and background. Similar to potency, our work does not consider *stealth*: While originally defined for opaque predicates [41], another obfuscation technique, we argue that any obfuscation technique can benefit from being hard to find for an attacker, increasing their workload to first finding the obfuscated code before meaningfully being able to attack it. That said, our work's focus on individual handlers does not make any attempt at hiding the obfuscated code but leaves this up to future work.

Limited prototype. Our current prototype serves more as a proof-of-concept than providing a tool industry could adopt. Most crucially, it currently does not support the application of obfuscation techniques across function or basic block boundaries but instead relies on LLVM's optimization passes to unroll code prior to processing. Similarly, memory accesses are implemented via a single handler, which features no particular protection. Both these decisions have been made to limit the engineering effort and could be implemented to improve our prototype. Although, we stress that it is not our intention to provide an industry-grade obfuscator for public use. With our scope on protecting individual core semantics, or handlers, rather than the entire VM, the whole VM structure itself is simple in nature and features little protection, for example, we use no bytecode blinding [23]. As mentioned above, this also includes the fact that we made no attempt at hiding the code, but our threat model assumes that an attacker has already successfully bypassed all these orthogonal protection mechanisms.

3.3.4 Future Work

Besides the limitations of our work, future work could tackle the following avenues.

Extension to more techniques. Future work could foremost further harden code obfuscation by considering and properly evaluating other obfuscation techniques, such as opaque predicates [40, 195] or range dividers [15], and orthogonal countermeasures, including self-modifying code [101] or anti-debugging techniques [34, 35, 66]. A thorough systematization and evaluation of available techniques would allow for the sustainable identification of strong ones, and it would permit to identify techniques exhibiting beneficial synergy effects when combined.

Evaluation metrics & practices. Fundamentally, obfuscation is security by obscurity, a principle long considered obsolete in fields such as cryptography [142]. Still, in this particular scenario, where provable security incurring reasonable overhead is difficult, if not impossible, to achieve, raising the bar for an attacker is often sufficient to protect intellectual property, making obfuscation a viable and widely used practice in the industry. In any case, the lack of provable properties makes a definitive evaluation challenging. Other than for fields such as fuzzing, where FuzzBench [120] provides a benchmarking platform, no standardized means of benchmarking or metrics exist for obfuscation schemes. Finding adequate metrics and deriving a benchmarking platform could be an interesting avenue for future research. A first step would be focusing on resilience, where we expect benchmarking evaluations to be reasonable. In contrast, more work is required o find suitable metrics to measure potency.

Conclusions

In this thesis, we have studied different aspects of the *automation of program analyses*. First, we have surveyed how *reproducible* existing research works in the field of fuzzing are. Without reproducibility, other research cannot build on specific works, nor can the industry adopt a technique with unclear benefits. Our findings indicate that many works have ample room for improvement. In particular, existing guidelines ensuring common grounds and a sane evaluation setup are often not followed, results are not statistically evaluated, and misuse of CVEs often inflates the practical impact. However, providing reproducible bug-finding tools is a crucial predecessor to a successful and sustainable automation of the bug-finding pipeline.

In a second step, we have discussed a new approach towards automatically generating exploits. Representing the last step in the bug pipeline, the technique achieves high flexibility: It allows us to encode arbitrary constraints by relaying the problem of finding a gadget chain to an SMT solver. At the same time, our approach can prove that no gadget chain can exist for a given set of pre- and postconditions, contrasting existing works in this realm. While the opportunity for misuse exists, automatically generating proof-of-concept exploits can help assess the threat arising from a particular bug. Thus, it can allow maintainers to prioritize fixing specific bugs over others.

Having studied the first and last step of the bug pipeline, for which automation provides a net positive impact, we then focus on a scenario where automation is predominantly used in undesired ways, namely the field of code (de-)obfuscation. Here, intellectual property is protected by making code hard to analyze, working on a security-by-obscurity basis. As our work underlines, existing approaches fail to live up to expec-

tations, as automated analysis techniques succeed in simplifying the protected code, making it amenable to further analysis. Studying the limits of the used analysis techniques, we designed a set of obfuscation techniques in the context of VM-based obfuscation that exploit inherent weaknesses of used techniques and, in combination, help thwart automated deobfuscation attacks. This shows that considering adjacent fields is important when improving automated analyses, but it also demonstrates that such analyses can be significantly hampered if desired.

Future research directions. Given the steady growth of software and its ever-increasing impact on our lives, the automation of program analyses will become an even more pressing issue. A natural goal for the near future is an automated approach that combines individual steps of the bug pipeline, from finding some input that causes abnormal behavior to reasoning on the root cause and capabilities of this bug to generating an exploit or providing a patch that mitigates the undesired behavior. With AIxCC [46] on the horizon, the arrival of first attempts in this direction that work for real-world scenarios may be imminent. The competition's focus on artificial intelligence as a substitute for human behavior represents a renewed belief that incredible recent advancements in the field of machine learning (leading to tools such as ChatGPT [135]) can be harnessed to overcome technical gaps that currently must be filled by human experts. It remains to be seen whether individual components in this pipeline, each having different requirements and making different assumptions, can be fused together solely by *artificial* intelligence to form a working pipeline that succeeds in analyzing and securing real-world applications.

That said, the immense diversity in hardware, software, and types of security vulnerabilities will make a "one-shot" solution challenging, requiring significant further research on individual, directed improvements. For example, when focusing on the field of fuzzing, we find several limiting factors, including the fuzzer's capability to *sense* a bug, where introducing new bug oracles would extend its capabilities to other bug types, or the quality of underlying fuzzing harnesses. Currently, human experts write such harnesses, leading to incompleteness, decay over time (if not properly maintained), and a general oversight of attack surface [180]. Automatically deriving a harness tailored toward the target-under-test, combined with a fuzzing technique optimized depending on the target's needs will significantly ease the adoption of fuzzing by non-experts and allow for further scaling. There is even more to do for automated exploit generation: To

the best of our knowledge, existing approaches are highly limited to specific scenarios or require strong assumptions, leaving most of the exploit generation up to a human expert. In particular, no generic end-to-end exploitation approach exists. With the developer's need for bug prioritization in mind, which will only increase with broader adoption and application of bug-finding techniques such as fuzzing, we have to devise more and more usable technologies to assess and convey the potential security impact of bugs. Here, academia will likely shift towards extracting more details from the initial crashing inputs to better showcase the context and capabilities of a bug and work towards automatically assessing these bugs. Another natural direction for this combination of bug finding and assessment is a better inclusion into Continuous Integration (CI) to enable analysis of patches and smaller snippets of code rather than testing a whole project once in an ad-hoc manner. Finally, software protection, in particular code obfuscation, research needs to keep up with the increasingly powerful deobfuscation techniques proposed. A promising step in this direction is the even tighter interlocking of different techniques to reduce the attack surface in terms of clearly separable pieces that are easy to attack. Similar to this work, we can expect future techniques to rely on inherent weaknesses of analyses to achieve lasting resilience. In more concrete terms, we anticipate more complex VM-based approaches that not only harden individual handlers but also make it difficult for analysts to identify interesting data flows. At the same time, future work in the domain of automated deobfuscation will increasingly turn towards combining existing techniques to benefit from synergy effects [57]. This will be combined with technical improvements: For example, synthesis, one of the most powerful simplification primitives, currently struggles with large constants or exhibits performance plateaus [107]. Overall, we can expect that the arms race between automatically simplifying obfuscated code and hardening it against new or stronger techniques will continue for the foreseeable future.

Across the board, automated program analyses will become even more important than they are right now, making it paramount to set the right foundation for reproducibility, thus enabling industry adoption and advances in research. At the same time, we must ensure our automation provides flexibility for a wide range of scenarios, or we jeopardize their broad applicability without human intervention. Requiring a high level of flexibility also ensures that we can use techniques in different contexts, such as when we turn an approach to generate exploits automatically into one that can help

developers with an initial, automated assessment of a bug's severity and exploitability. Finally, we must be aware that improving analysis techniques also comes with risks in adjacent fields. As we have seen, these risks can be mitigated to minimize the impact of nefarious use of automated deobfuscation, but this requires research on its own and awareness in the first place.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] Martin Abadi and Roger Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[3] AFL++ Team. CmpLog. `https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.cmplog.md`, 2020.

[4] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49):14–16, 1996.

[5] Allen Institute for AI. Semantic Scholar. `https://www.semanticscholar.org/`.

[6] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling Enumerative Program Synthesis via Divide and Conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2017.

[7] angr team. angrop. `https://github.com/angr/angrop`.

[8] Mohammad Rifat Arefin, Suraj Shetiya, Zili Wang, and Christoph Csallner. Fast Deterministic Black-box Context-free Grammar Inference. In *International Conference on Software Engineering (ICSE)*, 2024.

[9] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and Don'ts of Machine Learning in Computer Security. In *USENIX Security Symposium*, 2022.

[10] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed System Security (NDSS) Symposium*, 2019.

[11] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Network and Distributed System Security (NDSS) Symposium*, 2019.

[12] Association for Computing Machinery. Artifact Review and Badging Version 1.1. `https://www.acm.org/publications/policies/artifact-review-and-badging-current`, 2020.

[13] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Network and Distributed System Security (NDSS) Symposium*, 2011.

[14] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[15] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code Obfuscation against Symbolic Execution Attacks. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[16] Sebastian Banescu and Alexander Pretschner. A Tutorial on Software Obfuscation. *Advances in Computers*, 108:283–353, 2018.

[17] Sebastian Banescu, Samuel Valenzuela, Marius Guggenmos, Mohsen Ahmadvand, and Alexander Pretschner. Dynamic Taint Analysis versus Obfuscated Self-Checking. In *Annual Computer Security Applications Conference (ACSAC)*, 2021.

[18] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *USENIX Security Symposium*, 2023.

[19] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. "Synthesizing Input Grammars": A Replication Study. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[20] Gareth Bennett, Tracy Hall, and David Bowes. Some Automatically Generated Patches are more likely to be Correct than Others: an Analysis of Defects4J Patch Features. In *International Workshop on Automated Program Repair*, 2022.

[21] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[22] Tim Blazytko, Cornelius Aschermann, Moritz Schloegel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019.

[23] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *USENIX Security Symposium*, 2017.

[24] Tim Blazytko, Moritz Schloegel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. Aurora: Statistical Crash Analysis for Automated Root Cause Explanation. In *USENIX Security Symposium*, 2020.

[25] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented Programming: A new Class of Code-reuse Attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[27] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[28] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[29] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[30] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[31] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[32] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Network and Distributed System Security (NDSS) Symposium*, 2018.

[33] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[34] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. Advanced or Not? A Comparative Study of the Use of Anti-Debugging and Anti-VM Techniques in Generic and Targeted Malware. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2016.

[35] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Conference on Dependable Systems and Networks (DSN)*, 2008.

[36] Christian Collberg. The Tigress C Diversifier/Obfuscator. `http://tigress.wtf/`.

[37] Christian Collberg, Jack Davidson, Roberto Giacobazzi, Yuan Xiang Gu, Amir Herzberg, and Fei-Yue Wang. Toward Digital Asset Protection. *IEEE Intelligent Systems*, 26(6):8–13, 2011.

[38] Christian Collberg and Todd A Proebsting. Repeatability in Computer Systems Research. *Communications of the ACM (CACM)*, 59(3):62–69, 2016.

[39] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[40] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[41] Christian S. Collberg and Clark Thomborson. Watermarking, Tamper-proofing, and Obfuscation-tools for Software Protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[42] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stack-Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[43] Sebastian Danicic and Michael R. Laurence. Static Backward Slicing of Non-Deterministic Programs and Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(3):11:1–11:46, 2018.

[44] DARPA. DARPA Cyber Grand Challenge. `https://github.com/CyberGrandChallenge`, 2018.

[45] Robin David, Luigi Coniglioi, and Mariano Ceccato. QSynth – A Program Synthesis based Approach for Binary Code Deobfuscation. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2020.

[46] Defense Advanced Research Projects Agency (DARPA). Artificial Intelligence Cyber Challenge (AIxCC). `https://aicyberchallenge.com/`, 2023.

[47] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. Reproducibility and Replicability of Web Measurement Studies. In *ACM Web Conference*, 2022.

[48] Denuvo Software Solutions GmbH. Denuvo Anti-Tamper. `http://www.denuvo.com`.

[49] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[50] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[51] Thomas Dullien. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2017.

[52] Abdelrahman Eid. Reverse Engineering Snapchat (Part I): Obfuscation Techniques. `https://hot3eed.github.io/snap_part1_obfuscations.html`.

[53] Clément Elbaz, Louis Rilling, and Christine Morin. Fighting N-Day Vulnerabilities with Automated CVSS Vector Prediction at Disclosure. In *International Conference on Availability, Reliability and Security (ARES)*, 2020.

[54] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2017.

[55] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage Binary Code Obfuscation using Improved Virtual Machine. In *International Conference on Information Security*, 2011.

[56] Matteo Favaro. Tickling VMProtect with LLVM. `https://secret.club/2021/09/08/vmprotect-llvm-lifting-1.html`, 2019.

[57] Matteo Favaro and Tim Blazytko. Improving MBA Deobfuscation using Equality Saturation. `https://secret.club/2022/08/08/eqsat-oracle-synthesis.html`, 2022.

[58] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[59] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. *ACM SIGPLAN Notices*, 52(6):422–436, 2017.

[60] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[61] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[62] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[63] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, , and Giovanni Vigna. ACTOR: Action-Guided Kernel Fuzzing. In *USENIX Security Symposium*, 2023.

[64] Forum of Incident Response and Security Teams, Inc. Common Vulnerability Scoring System SIG. `https://www.first.org/cvss/`, 2015.

[65] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A Bounded Model Checker for Smart Contracts. In *USENIX Security Symposium*, 2020.

[66] Michael N Gagnon, Stephen Taylor, and Anup K Ghosh. Software Protection through Anti-Debugging. *IEEE Security & Privacy*, 5(3):82–84, 2007.

[67] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based Directed Whitebox Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009.

[68] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic Software Repair: A Survey. In *International Conference on Software Engineering (ICSE)*, 2018.

[69] GCC Team. GCC's Optimize Options. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[70] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM (CACM)*, 55(3):40–44, 2012.

[71] Patrice Godefroid and Ankur Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. *ACM SIGPLAN Notices*, 47(6):441–452, 2012.

[72] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software. `https://github.com/google/oss-fuzz`.

[73] Google. DynamoRIO. `https://dynamorio.org/`, 2010.

[74] Google. Fuzzer-Test-Suite. `https://github.com/google/fuzzer-test-suite`, 2016.

[75] Google Project Zero. CompareCoverage. `https://github.com/googleprojectzero/CompareCoverage`, 2019.

[76] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Mining Input Grammars from Dynamic Control Flow. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[77] Rahul Gopinath, Hamed Nemati, and Andreas Zeller. Input Algebras. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2021.

[78] Sumit Gulwani. Dimensions in Program Synthesis. In *International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2010.

[79] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

[80] Sumit Gulwani and Nebojsa Jojic. Program Verification as Probabilistic Inference. *ACM SIGPLAN Notices*, 42(1):277–289, 2007.

[81] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. AntiFuzz: Impeding Fuzzing Audits of Binary Executables. In *USENIX Security Symposium*, 2019.

[82] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, and Thorsten Holz. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *USENIX Security Symposium*, 2024.

[83] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. Learning to Predict Severity of Software Vulnerability Using Only Vulnerability Description. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

[84] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W Davidson. Zipr: Efficient Static Binary Rewriting for Security. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

[85] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A Ground-Truth Fuzzing Benchmark. *ACM on Measurement and Analysis of Computing Systems (POMACS)*, 4(3):49:1–49:29, 2020.

[86] Sean Heelan. *Greybox Automatic Exploit Generation for Heap Overflows in Language Interpreters*. PhD thesis, University of Oxford, 2020.

[87] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic Heap Layout Manipulation for Exploitation. In *USENIX Security Symposium*, 2018.

[88] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[89] Adrian Herrera, Mathias Payer, and Antony L. Hosking. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 32(5), 2023.

[90] Marc Heuse, Dominik Maier, Andrea Fioraldi, and Heiko Eissfeldt. American Fuzzy Lop plus plus (AFL++). `https://github.com/AFLplusplus/AFLplusplus`.

[91] Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, and Jack Davidson. Zipr++ Exceptional Binary Rewriting. In *Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[92] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[93] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[94] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-only Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[95] Christopher Stanislaw Jelesnianski. *Practical Exploit Mitigation Design Against Code Re-Use and System Call Abuse Attacks*. PhD thesis, Virginia Tech, 2023.

[96] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. Evaluating and Improving Hybrid Fuzzing. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2023.

[97] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. Evocatio: Conjuring Bug Capabilities from a Single PoC. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[98] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-Fuzzing Techniques. In *USENIX Security Symposium*, 2019.

[99] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. VMAttack: Deobfuscating Virtualization-based Packed Binaries. In *International Conference on Availability, Reliability and Security (ARES)*, 2017.

[100] Hong Jin Kang, Khai Loong Aw, and David Lo. Detecting False Alarms from Automatic Static Analysis Tools: How far are we? In *International Conference on Software Engineering (ICSE)*, 2022.

[101] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. Exploiting Self-modification Mechanism for Program Protection. In *IEEE Signature Conference on Computers, Software, and Applications (COMPSAC)*, 2003.

[102] Gal Katz and Doron Peled. Genetic Programming and Model Checking: Synthesizing new Mutual Exclusion Algorithms. In *International Symposium on Automated Technology for Verification and Analysis*, 2008.

[103] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In *USENIX Security Symposium*, 2023.

[104] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[105] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[106] Patrick Kochberger, Sebastian Schrittwieser, Stefan Schweighofer, Peter Kieseberg, and Edgar Weippl. SoK: Automatic Deobfuscation of Virtualization-protected Applications. In *International Conference on Availability, Reliability and Security (ARES)*, 2021.

[107] Jason R Koenig, Oded Padon, and Alex Aiken. Adaptive Restarts for Stochastic Synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[108] Jaehyung Lee and Woosuk Lee. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[109] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. Deobfuscation of Virtualization-obfuscated Code through Symbolic Execution and Compilation Optimization. In *International Conference on Information and Communications Security*, 2018.

[110] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[111] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazel-

wood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[112] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, 2019.

[113] Valentin J. M. 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*, 47(11):2312–2331, 2019.

[114] David E Mann and Steven M Christey. Towards a Common Enumeration of Vulnerabilities. In *Workshop on Research with Security Vulnerability Databases*, 1999.

[115] Zohar Manna and Richard Waldinger. Synthesis: Dreams → Programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, 1979.

[116] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic Repair of Real Bugs in Java: A Large-scale Experiment on the Defects4J Dataset. *Empirical Software Engineering*, 22:1936–1964, 2017.

[117] Nate Mathews, James K Holland, Se Eun Oh, Mohammad Saidur Rahman, Nicholas Hopper, and Matthew Wright. SoK: A Critical Evaluation of Efficient Website Fingerprinting Defenses. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[118] Michael Matz. Comment on CVEs. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=87675#c1`, 2018.

[119] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[120] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[121] Boyan Milanov. ROPium. `https://github.com/Boyan-MILANOV/ropium`.

[122] Barton P. Miller, Lars Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM (CACM)*, 33(12):32–44, 1990.

[123] MITRE Corporation. CVE® Program Mission. `https://www.cve.org/`, 1999.

[124] Martin Monperrus. Automatic Software Repair: A Bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.

[125] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Repositioning of Static Analysis Alarms. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[126] Stefan Nagy and Matthew Hicks. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[127] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing. In *USENIX Security Symposium*, 2021.

[128] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[129] National Institute of Standards and Technology. National Vulnerability Database. `https://nvd.nist.gov/`.

[130] Maria-Irina Nicolae, Max Eisele, and Andreas Zeller. Revisiting Neural Program Smoothing for Fuzzing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.

[131] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2015.

[132] Aina Niemetz and Mathias Preiner. Bitwuzla. In *International Conference on Computer Aided Verification*, 2023.

[133] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):53–58, 2014.

[134] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[135] OpenAI. Introducing ChatGPT. `https://openai.com/blog/chatgpt`, 2022.

[136] Oreans Technologies. Themida – Advanced Windows Software Protection System. `https://www.oreans.com/Themida.php`.

[137] Lin Padgham, Young Lee, Shazia Sadiq, Michael Winikoff, Alan Fekete, Stephen MacDonell, Dali Kaafar, and Stefanie Zollmann. CORE Rankings. `https://www.core.edu.au/conference-portal`.

[138] Nikhil Parasaram, Earl T Barr, Sergey Mechtaev, and Marcel Böhme. Precise Data-Driven Approximation for Program Analysis via Fuzzing. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2023.

[139] Younggi Park, Hwiwon Lee, Jinho Jung, Hyungjoon Koo, and Huy Kang Kim. BENZENE: A Practical Root Cause Analysis System with an Under-Constrained State Mutation. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[140] PaX Team. PaX NOEXEC Design & Implementation. `http://pax.grsecurity.net/docs/noexec.txt`.

[141] PaX Team. Address Space Layout Randomization (ASLR). `https://pax.grsecurity.net/docs/aslr.txt`, 2001.

[142] Fabien AP Petitcolas. Kerckhoffs' Principle. In *Encyclopedia of Cryptography, Security and Privacy*, pages 1–2. Springer, 2023.

[143] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *USENIX Security Symposium*, 2020.

[144] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2015.

[145] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security (NDSS) Symposium*, 2017.

[146] Benjamin Reichenwallner and Peter Meerwald-Stadler. Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions. In *ACM Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*, 2022.

[147] Benjamin Reichenwallner and Peter Meerwald-Stadler. Simplification of General Mixed Boolean-Arithmetic Expressions: GAMBA. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2023.

[148] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[149] Rolf Rolles. Unpacking Virtualization Obfuscators. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[150] Rust Team. Rust. `https://www.rust-lang.org/`, 2014.

[151] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-Level Fuzzing. In *USENIX Security Symposium*, 2021.

[152] Jonathan Salwan. ROPgadget. `https://github.com/JonathanSalwan/ROPgadget`.

[153] Jonathan Salwan. VMProtect Devirtualization. `https://github.com/JonathanSalwan/VMProtect-devirtualization`, 2022.

[154] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From Virtualized Code back to the Original. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.

[155] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *USENIX Security Symposium*, 2022.

[156] Nico Schiller, Merlin Chlosta, Moritz Schloegel, Nils Bars, Thorsten Eisenhofer, Tobias Scharnowski, Felix Domke, Lea Schönherr, and Thorsten Holz. Drone Security and the Mysterious Case of DJI's DroneID. In *Network and Distributed System Security (NDSS) Symposium*, 2023.

[157] Sascha Schirra. Ropper. `https://github.com/sashs/Ropper`.

[158] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[159] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting Software through Obfuscation: Can it Keep Pace with Progress in Code Analysis? *ACM Computing Surveys (CSUR)*, 49(1):1–37, 2016.

[160] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[161] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[162] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[163] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[164] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*, 2011.

[165] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari, and Stephanie M Schwartz. A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[166] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[167] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[168] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy (S&P)*, 2009.

[169] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SoK: (State of) the Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[170] Shivam Shrirao. Can we Bruteforce ASLR? `https://ret2rop.blogspot.com/2018/08/can-we-bruteforce-aslr.html`, 2018.

[171] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[172] Solar Designer. Return-to-libc Attack. *Bugtraq, Aug*, 1997.

[173] Sony DADC. SecuROM Software Protection. https://www2.securom.com/Digital-Rights-Management.68.0.html.

[174] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. Template-based Program Verification and Program Synthesis. *International Journal on Software Tools for Technology Transfer*, 15:497–518, 2013.

[175] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[176] 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 *Network and Distributed System Security (NDSS) Symposium*, 2016.

[177] Aaron Stump, Clark W Barrett, David L Dill, and Jeremy Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *IEEE Symposium on Logic in Computer Science*, 2001.

[178] Robert Swiecki. Honggfuzz. https://github.com/google/honggfuzz, 2015.

[179] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[180] Alexander Tarasikov. Why CVE-2022-3602 was not Detected by Fuzz Testing. https://allsoftwaresucks.blogspot.com/2022/11/why-cve-2022-3602-was-not-detected-by.html, 2023.

[181] The Paradyn Project. Dyninst. https://github.com/dyninst/dyninst, 1993.

[182] Unknown Authors. LAF-Intel: Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/, 2016.

[183] Arjan van de Ven. ExecShield. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[184] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. SoK: Benchmarking Flaws in Systems Security. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[185] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT Solvers in Software Security. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[186] Martin Vechev and Eran Yahav. Deriving Linearizable Fine-grained Concurrent Objects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[187] VMProtect Software. VMProtect Software. `https://vmpsoft.com/`.

[188] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[189] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware Greybox Fuzzing. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2019.

[190] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[191] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards Automated Heap Feng Shui. In *USENIX Security Symposium*, 2021.

[192] Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE)*, 1981.

[193] Johannes Willbold. Efficient Fuzzing of VM-obfuscated Code. Master's thesis, Ruhr-Universität Bochum, 2020.

[194] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partially-virtualized Binary Code Simplification. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[195] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. Manufacturing Resilient Bi-Opaque Predicates against Symbolic Execution. In *Conference on Dependable Systems and Networks (DSN)*, 2018.

[196] Babak Yadegari and Saumya Debray. Bit-level Taint Analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2014.

[197] Babak Yadegari and Saumya Debray. Symbolic Execution of Obfuscated Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[198] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[199] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[200] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *USENIX Security Symposium*, 2021.

[201] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[202] Michał Zalewski. American Fuzzy Lop. `https://lcamtuf.coredump.cx/afl/`, 2013.

[203] Michał Zalewski. AFL-Fuzz: Crash Exploration Mode. `https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html`, 2014.

[204] Andreas Zeller. Program Analysis: A Hierarchy. In *ICSE Workshop on Dynamic Analysis (WODA)*, 2003.

[205] Bin Zhang, Jiongyi Chen, Runhao Li, Chao Feng, Ruilin Li, and Chaojing Tang. Automated Exploitable Heap Layout Generation for Heap Overflows Through Manipulation Distance-Guided Fuzzing. In *USENIX Security Symposium*, 2023.

[206] Naiqian Zhang, Daroc Alden, Dongpeng Xu, Shuai Wang, Trent Jaeger, and Wheeler Ruml. No Free Lunch: On the Increased Code Reuse Attack Surface of Obfuscated Programs. In *Conference on Dependable Systems and Networks (DSN)*, 2023.

[207] Quanjun Zhang, Yuan Zhao, Weisong Sun, Chunrong Fang, Ziyuan Wang, and Lingming Zhang. Program Repair: Automated vs. Manual. *arXiv preprint arXiv:2203.05166*, 2022.

[208] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *International Workshop on Information Security Applications*, 2007.

Part II

Publications

List of Publications

This thesis is based on the three papers that are referenced below and included verbatim in Appendices A to C, respectively. The next pages list all of the author's 17 additional contributions.

Publications in this Thesis

[1] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[2] Moritz Schloegel, Tim Blazytko, Julius Basler, Fabian Hemmer, and Thorsten Holz. Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains. In *European Symposium on Research in Computer Security (ESORICS)*, 2021.

[3] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening Code Obfuscation Against Automated Attacks. In *USENIX Security Symposium*, 2022.

Other Contributions

[1] Sebastian Wallat, Marc Fyrbiak, Moritz Schloegel, and Christof Paar. A Look at the Dark Side of Hardware Reverse Engineering – A Case Study. In *IEEE International Verification and Security Workshop (IVSW)*, 2017.

[2] Tim Blazytko, Cornelius Aschermann, Moritz Schloegel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019.

[3] Tim Blazytko, Moritz Schloegel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. Aurora: Statistical Crash Analysis for Automated Root Cause Explanation. In *USENIX Security Symposium*, 2020.

[4] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *USENIX Security Symposium*, 2022.

[5] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-Picking: Differential Fuzzing of JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[6] Nico Schiller, Merlin Chlosta, Moritz Schloegel, Nils Bars, Thorsten Eisenhofer, Tobias Scharnowski, Felix Domke, Lea Schönherr, and Thorsten Holz. Drone Security and the Mysterious Case of DJI's DroneID. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[7] Johannes Willbold, Moritz Schloegel, Manuel Vögele, Maximilian Gerhardt, Thorsten Holz, and Ali Abbasi. Space Odyssey: An Experimental Software Security Analysis of Satellites. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[8] Nico Schiller, Xinyi Xu, Lukas Bernhard, Nils Bars, Moritz Schloegel, and Thorsten Holz. Novelty Not Found: Adaptive Fuzzer Restarts to Improve Input Space Coverage (Registered Report). In *International Fuzzing Workshop (FUZZING)*, 2023.

[9] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *USENIX Security Symposium*, 2023.

[10] Tobias Scharnowski, Simon Wörner, Felix Buchmann, Nils Bars, Moritz Schloegel, and Thorsten Holz. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In *USENIX Security Symposium*, 2023.

[11] Daniel Klischies, Moritz Schloegel, Tobias Scharnowski, Mikhail Bogodukhov, David Rupprecht, and Veelasha Moonsamy. Instructions Unclear: Undefined Behavior in Cellular Network Specifications. In *USENIX Security Symposium*, 2023.

[12] Johannes Willbold, Moritz Schloegel, Florian Göhler, Tobias Scharnowski, Nils Bars, Simon Wörner, and Thorsten Holz. Scaling Software Security Analysis to Satellites: Automated Fuzz Testing and Its Unique Challenges. In *IEEE Aerospace Conference*, 2024.

[13] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *USENIX Security Symposium*, 2024.

[14] Joschua Schilling, Andreas Wendler, Philipp Görz, Nils Bars, Moritz Schloegel, and Thorsten Holz. A Binary-level Thread Sanitizer or Why Sanitizing on the Binary Level is Hard. In *USENIX Security Symposium*, 2024.

[15] Nico Schiller, Xinyi Xu, Lukas Bernhard, Nils Bars, Moritz Schloegel, and Thorsten Holz. Novelty Not Found: Adaptive Fuzzer Restarts to Improve Input Space Coverage. *Manuscript in Submission*, 2024.

[16] Johannes Willbold, Moritz Schloegel, Thorsten Holz, Martin Strohmeier, and Vincent Lenders. VSAsTer: Uncovering Inherent Security Issues in Current VSAT System Practices. *Manuscript in Submission*, 2024.

[17] Felix Weißberg, Jonas Möller, Tom Ganz, Lukas Pirch, Lukas Seidel, Moritz Schloegel, Thorsten Eisenhofer, and Konrad Rieck. SoK: Where to Fuzz? Assessing Target Selection Methods in Directed Fuzzing. *Manuscript in Submission*, 2024.

SoK: Prudent Evaluation Practices for Fuzzing

Publication Data

Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

Paper has received a *Distinguished Paper Award* at IEEE S&P.

Permission. *© 2024 IEEE. Reprinted, with permission, from the citation specified above. In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Ruhr University Bochum's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to* http://www.ieee.org/publications_standards/publications/rights/rights_link.html *to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.*

SoK: Prudent Evaluation Practices for Fuzzing

Moritz Schloegel[1], Nils Bars[1], Nico Schiller[1], Lukas Bernhard[1], Tobias Scharnowski[1], Addison Crump[1], Arash Ale Ebrahim[1], Nicolai Bissantz[2], Marius Muench[3], and Thorsten Holz[1]

[1] CISPA Helmholtz Center for Information Security
[2] Ruhr University Bochum
[3] University of Birmingham

Abstract

Fuzzing has proven to be a highly effective approach to uncover software bugs over the past decade. After AFL popularized the groundbreaking concept of lightweight coverage feedback, the field of fuzzing has seen a vast amount of scientific work proposing new techniques, improving methodological aspects of existing strategies, or porting existing methods to new domains. All such work must demonstrate its merit by showing its applicability to a problem, measuring its performance, and often showing its superiority over existing works in a thorough, empirical evaluation. Yet, fuzzing is highly sensitive to its target, environment, and circumstances, e. g., randomness in the testing process. After all, relying on randomness is one of the core principles of fuzzing, governing many aspects of a fuzzer's behavior. Combined with the often highly difficult to control environment, the *reproducibility* of experiments is a crucial concern and requires a prudent evaluation setup. To address these threats to validity, several works, most notably *Evaluating Fuzz Testing* by Klees et al., have outlined how a carefully designed evaluation setup should be implemented, but it remains unknown to what extent their recommendations have been adopted in practice.

In this work, we systematically analyze the evaluation of 150 fuzzing papers published at the top venues between 2018 and 2023. We study how existing guidelines are implemented and observe potential shortcomings and pitfalls. We find a surprising disregard of the existing guidelines regarding statistical tests and systematic errors in fuzzing evaluations. For example, when investigating reported bugs, we find that the search for vulnerabilities in real-world software leads to authors requesting and receiving CVEs of questionable quality. Extending our literature analysis to the practical

domain, we attempt to reproduce claims of eight fuzzing papers. These case studies allow us to assess the practical reproducibility of fuzzing research and identify archetypal pitfalls in the evaluation design. Unfortunately, our reproduced results reveal several deficiencies in the studied papers, and we are unable to fully support and reproduce the respective claims. To help the field of fuzzing move toward a scientifically reproducible evaluation strategy, we propose updated guidelines for conducting a fuzzing evaluation that future work should follow.

1 Introduction

Fuzzing, a portmanteau of "fuzz testing", has gained much attention in recent years, and the method has proven to be highly successful in uncovering many types of faults in software systems. Companies such as Meta, Google, and Oracle have invested significant resources in this technology and use it to test their products. Large software projects such as web browsers or the Linux kernel incorporate fuzzing into their development cycle, and Google is running an extensive and continuous fuzzing campaign for more than 1,200 open-source projects via OSS-Fuzz [1]. Beyond the wide acceptance in the industry, a large number of academic papers have proposed numerous improvements and novel techniques to enhance fuzzing further. More specifically, we found that, over the past six years, more than 280 papers on fuzzing have been published in the top computer security and software engineering venues.

A cornerstone of fuzzing research, and science in general, is that other researchers can critically assess the correctness of scientific results. To this end, the research results must be *reproducible*, meaning that another group should be able to obtain the same results using the same experimental setup, often by using a research artifact provided by the authors [2]. Reproducibility is paramount for other researchers to understand, trust, and build on the research results.

To enable high-quality research and provide a common foundation for evaluating fuzzing methods, several works describe how newly proposed fuzzing approaches should be evaluated. In 2018, the first and most influential paper describing a reproducible evaluation design was published by Klees et al. [3]. It describes guidelines to advise researchers on how fuzzing research should evaluate their respective contributions. For example, a crucial insight introduced by Klees et al. is the repetition of experiments to account for the inherent randomness of the fuzzing process. Although Klees et al. recommend "a sufficient number of trials" and use 30 trials in their own experiments, we found that in practice, this recommendation is interpreted as anything between three and 20 repetitions. Another guideline is to confirm the fuzzers' performance statistically; however, this makes little sense with few repetitions and is often skipped.

In this work, we systematically review how the recommendations for evaluating fuzzing methods are implemented in practice and critically evaluate the reproducibility of fuzzing research. We propose revised best practices for evaluating fuzzing methods and point out pitfalls that we have observed in practice. In other fields, such work

has had a significant impact on improving research from a methodological point of view [4, 5, 6, 7].

We conduct a thorough literature review of 150 fuzzing papers published in prestigious A* venues—as ranked by CORE2023 [8]—between 2018 and 2023. While we primarily focus on computer security venues, namely IEEE Symposium on Security and Privacy (S&P), USENIX Security Symposium (USENIX), ACM Conference on Computer and Communications Security (CCS), and ISOC Network and Distributed System Security (NDSS) Symposium, we also examine three software engineering venues: IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), and International Conference on Software Engineering (ICSE). For all papers, we: (i) systematically analyze how evaluations are conducted (in terms of metrics, targets, baselines, reported bugs, etc.), (ii) check whether common fuzzing guidelines (as outlined by Klees et al. [3] or embodied in implicit community wisdom, e. g., "do not use artificial bug datasets") are followed, and (iii) investigate potential flaws threatening the validity of the respective evaluation.

Following our literature analysis, we present eight case studies of fuzzing papers across different fields and attempt to reproduce (parts of) their evaluation. For each case study, we discuss any shortcomings we have identified because they illustrate potential pitfalls of which researchers should be aware. Note that these case studies are *not* intended to point fingers or criticize any particular work. Instead, we aim to highlight potential challenges that can affect the outcome of a research paper and explore what aspects need to be considered when designing the evaluation of a fuzzing method. Based on the findings of our literature review and case studies, we propose best practices for evaluating future fuzzing methods to enable reproducible research.

In summary, we make the following key contributions:

- We conduct a systematic literature survey of 150 papers published in the past six years at top venues to assess how fuzzing methods are typically evaluated.

- We attempt to reproduce eight papers to assess the practical aspect of fuzzing evaluations. In doing so, we identify several obstacles that illustrate (sometimes subtle) shortcomings of evaluating fuzzing methods.

- Based on our lessons learned, we provide revised recommendations and best practices for future fuzzing evaluations.

Supplementary material for this work is available online at `https://github.com/fuzz-evaluator/`, including our reproduction artifacts and recommended best practices for future work (see `https://github.com/fuzz-evaluator/guidelines`).

2 Fuzzing Evaluation Guidelines

We first provide a brief overview of fuzzing before describing several generally accepted best practices that guide a typical fuzzing evaluation.

2.1 Background on Fuzzing

Fuzzing, also referred to as *fuzz testing*, is a dynamic testing technique with the goal of uncovering bugs in systems. This typically happens by mutating some initial input(s) to the system or by deriving inputs from input specifications such as grammars. While processing the provided input, the system under test is monitored for *interesting* behavior. Beyond easily observable faults, such as program crashes, fuzzers can use more sophisticated bug oracles, such as sanitizers or differential testing. Moreover, modern fuzzers often use lightweight instrumentation to receive coverage feedback, allowing them to track inputs that executed previously unseen edges. A comprehensive overview of various fuzzing techniques can be found in the *Fuzzing Book* [9], and several surveys present a comprehensive overview of this topic [10, 11] or open challenges in this domain [12]. Most fuzzing research proposes an improvement by way of new techniques, new components, or entirely new fuzzers—few works focus on the theory behind fuzzing [13, 14, 15, 16].

A fundamental principle of all fuzzers is the inherent inclusion of randomness into the testing process. Starting from the scheduling order of the process, through the input and the mutations applied to it, to the fuzzing environment (including functions such as `getpid`, `time`, or `rand`, or shared resources such as the filesystem), there are numerous sources of randomness that make deterministic and reproducible execution challenging.

2.2 Guidelines of *Evaluating Fuzz Testing*

The randomized nature of fuzzing needs to be taken into account during the evaluation, which leads to challenges with reproducibility of research results in practice. Hence, the seminal paper by Klees et al. [3] outlined several guidelines on how a proper fuzzing evaluation should be conducted. For a reproducible and fair evaluation, they propose the following recommendations:

Recommendation 1 – Baseline: A comparison with a relevant and reasonable baseline is imperative to show what improvement a particular fuzzer provides.

Recommendation 2 – Targets: A relevant sample of targets to compare against is necessary. This includes benchmark programs with known bugs that can be used as a ground truth to measure bug detection capabilities.

Recommendation 3 – Setup & Parameters: Due to the inherent randomness of fuzzing, individual runs with the same configuration can yield significantly different outcomes. To address this problem, Klees et al. propose repeating the experiment multiple times. Similarly, fuzzing performance may vary within a single run, so short runtimes are not appropriate for extrapolating the behavior of a fuzzer over longer times. They propose 24 hours as a reasonable fuzzer runtime and recommend plotting the performance over time. Seed sets must be well documented and carefully selected; ideally, various sets, including the empty or uninformed seed, are tested.

Recommendation 4 – Evaluation Metrics: Ideally, fuzzing evaluations should not be based on proxy metrics such as code coverage alone, but on a fuzzer's ability to find bugs, i. e., the goal for which it was designed. In particular, an evaluation should not rely on heuristics such as AFL's coverage profiles or stack hashing. Complementing the evaluation on bug detection, Klees et al. recommend code coverage in terms of basic blocks or edges as secondary metric.

Recommendation 5 – Statistical Evaluation: Finally, the fuzzing evaluation should undergo statistical evaluation to rule out that the observed behavior is by mere chance. This requires a *sufficient* number of trials (Klees et al. themselves use 30); then, a statistical test such as the Mann-Whitney U-test or bootstrap-based methods should be used to test the null hypothesis that the new method exhibits no difference compared to a reasonable baseline.

2.3 Guidelines of *FuzzBench*

FuzzBench [17], a benchmarking suite for general-purpose fuzzer evaluation developed by Google, provides several target programs and aims to provide a standardized setup for fair comparison of fuzzers. FuzzBench is the successor to the Google Fuzzer Test Suite (FTS) [18]. During their extensive evaluation, the authors made two key observations that can serve as a guideline for future fuzzing research. First, the performance of a fuzzer varies significantly depending on the number of initial seeds; running without seeds allows for studying the difference when only a particular fuzzer can solve some comparisons/branches. Second, using a saturated corpus for fuzzing is *not* recommended, as fuzzers are barely capable of augmenting it. Even though this is common in practice, it is not well suited to discern or measure the performance of fuzzers.

2.4 Guidelines of *On the Reliability of Coverage*

More recently, Böhme et al. [15] made a number of recommendations based on their evaluation of the reliability of coverage. In particular, they recommend to use at least ten representative programs, each tested at least ten times for at least 12 hours (preferably, each value should be doubled). The selected programs should be real-world programs, and a bug evaluation should be done on real-world bugs. In addition to bugs, code coverage should also be evaluated—both using established metrics. In particular, fuzzer-specific measures such as AFL's unique paths should be avoided. For comparison, authors should choose a suitable baseline, such as the fuzzer on top of which the new technique is implemented. Authors should consider splitting benchmarks into a *training* and *validation* set to avoid overfitting. To confirm evaluation results, authors must measure significance and effect size using established techniques. They should discuss threats to the validity of their evaluation and how they mitigated them. Finally, authors should carefully document their setup and publish evaluation artifacts on long-term stable platforms such as Zenodo.

2.5 Fuzzing Benchmarks

Over the years, several *standardized benchmarks* and *platforms* to conduct fair and comparable fuzzing evaluations have been proposed, e. g., Google's Fuzzer-Test-Suite [18] (2016; superseded by FuzzBench), LAVA-M [19] (2016), CGC [20] (2018), Magma [21]

(2020), FuzzBench [17] (2020), Unibench [22] (2021), ProFuzzBench [23] (2021), and RevBugBench [24] (2022).

These benchmark platforms aim to measure the performance of general-purpose fuzzing, except for ProFuzzBench, which focuses on stateful protocol fuzzing. Overall, we can distinguish between benchmarks focusing on the comparison of achieved coverage (Google's Fuzzer-Test-Suite, Unibench, FuzzBench, and ProFuzzBench) and those focusing on the bug-finding capabilities of the fuzzing technique (LAVA-M, CGC, Magma, and RevBugBench). In the latter category, some utilize artificial bug injection (LAVA-M and CGC), make efforts to port actual vulnerabilities to the latest version of a program (Magma), or to revert fixes (RevBugBench). Artificial bug injection methods often introduce shallow bugs that are amenable to fuzzers, and are generally no longer recommended for an evaluation [17, 24, 25, 26].

3 Literature Analysis

With these guidelines and benchmarks in mind, we now study their adoption to better understand what best practices are used in fuzzing research. To this end, we perform a comprehensive literature survey of recent fuzzing papers.

3.1 Method

We examine all fuzzing papers published at the top computer security and software engineering conferences between 2018 and 2023*. We include a paper in our analysis if its focus is on fuzzing, e. g., it proposes a new method or extensively evaluates existing ones. In contrast, we exclude papers using fuzzers as a means to support their primary focus, e. g., solely to generate some diverse inputs. We identify 289 candidate papers for which we collect metadata about the underlying evaluation method, including whether the paper successfully participated in an artifact evaluation process. We then randomly select 52% (150) from these 289 papers and manually review them, i. e., study the design and evaluation of the work in detail. Table 1 shows an overview of analyzed papers.

We investigate whether the fuzzing evaluation guidelines outlined in Section 2 are followed or whether an evaluation deviates from them. We want to stress that there

* For 2023, ASE and FSE have not published the papers at the time of writing. We therefore work with available preprints.

Table 1: Overview of analyzed papers. © 2024 IEEE

Year	Venue	Papers	Studied
2023	ASE*	[27], [28], [29]	3/7
	FSE*	[30]	1/6
	ICSE	[31], [32], [33], [34], [35]	5/11
	CCS	[36], [37], [38], [39]	4/9
	NDSS	[40], [41], [42]	3/4
	S&P	[43], [44], [45]	3/9
	USENIX	[46], [47], [48], [49], [50], [51],[52], [53], [54], [55], [56], [57]	12/29
2022	ASE	[58], [59]	2/4
	FSE	[60], [61]	2/6
	ICSE	[62], [63], [64], [65], [66], [67], [68], [69]	8/17
	CCS	[70], [71], [72], [73], [74], [75], [76]	7/8
	NDSS	[77], [78], [79]	3/6
	S&P	[80], [81], [82], [83], [84]	5/9
	USENIX	[85], [86], [24], [87], [88], [89], [90], [91], [92]	9/19
2021	ASE	[93], [94]	2/6
	FSE	[17], [95]	2/4
	ICSE	[96], [97], [98]	3/6
	CCS	[99], [100], [101], [102], [103], [104]	6/13
	NDSS	[105], [106], [107]	3/6
	S&P	[108], [109]	2/7
	USENIX	[110], [111], [112], [113]	4/13
2020	ASE	[114], [115]	2/4
	FSE	[116], [117], [118]	3/7
	ICSE	[119], [120], [121], [122]	4/6
	CCS	[123]	1/2
	NDSS	[124], [125], [26]	3/4
	S&P	[126], [127], [128], [129], [130], [131]	6/7
	USENIX	[132], [133], [134], [135], [136], [137], [138], [139], [140], [141], [142]	11/19
2019	ASE	–	0/0
	FSE	[143]	1/4
	ICSE	[144], [145], [146], [147], [148]	5/7
	CCS	[149], [150]	2/3
	NDSS	[151], [152], [153]	3/4
	S&P	[154], [155], [156], [157], [158]	5/5
	USENIX	[159], [160], [161], [162], [163], [164]	6/6
2018	ASE	[165]	1/2
	FSE	–	0/0
	ICSE	–	0/0
	CCS	[166]	1/2
	NDSS	[167], [168]	2/2
	S&P	[169], [170]	2/3
	USENIX	[171], [172], [173]	3/3
total #papers analyzed			150/289

* *limited to available preprints*

may be good reasons to deviate from these guidelines, making a manual review and judgment on a case-by-case basis mandatory. We also study whether the evaluations performed expose flaws that future fuzzing papers could avoid.

3.2 Results

We study the papers regarding their reproducibility, targets, fuzzers, evaluation setup in terms of resources, common metrics, and statistical evaluation.

3.2.1 Reproducibility

A crucial aspect of verifying and advancing science is the ability to reproduce existing research results. When examining the metadata we collected for all 289 fuzzing papers, we find that 74% (214) publish the code of their technique, while 23% (66) do not share their code. Some do not contribute new code, upstreamed their code, or have not yet released the code (applies to FSE, which will take place after time of writing). Regarding other data (excluding code), we find that 11% (31) share data, 20 of which publish data as a substitute because they do not share their code or have no code to share. All software engineering conferences (ASE, FSE, and ICSE), USENIX Security, and CCS (since 2023) offer an artifact evaluation process where independent reviewers assess the published research artifact (for 2023, ASE and FSE have not yet published this data). Our analysis found that 36% (103) of the papers did not have access to such an artifact evaluation; 37% (107) had access but opted to not participate or failed to receive any badge. Only 23% (66) of the papers have one or more badges. Of these, 64 are considered *available* and 63 *functional* or *reusable*, a crucial requirement for reproduction. USENIX Security and CCS offer to reproduce the results of a paper, which only 16 out of 57 eligible papers achieved. We emphasize that artifact evaluation has been introduced only in recent years, but participation is rising. CCS offered artifact evaluation for the first time in 2023, further supporting this trend.

> With 74%, a majority of works releases their code. Despite being relatively new, 60% of the papers already had access to artifact evaluation, with adoption lagging behind at 23% of papers that obtained a badge.

3.2.2 Targets under Test

To showcase the strengths of an approach, a suitable set of targets is required. Looking at the distribution of used targets (excluding datasets) in Table 2, we find that they are strongly biased towards byte-oriented file formats, especially binutils. On average, fuzzing papers evaluate on 8.9 targets. In summary, we found 753 different targets used across all studied papers; of these, 76% (576) were evaluated in only one paper. In addition to real-world targets, a common way of reproducibly measuring fuzzer performance is using benchmarks. Figure 1 shows how benchmarks have been adopted in the past years. In total, 61% (91) of the papers use no benchmark, 17% (26) use LAVA-M [19], 10% (15) use FuzzBench [17], 8% (12) use Google's Fuzzer Test Suite (FTS) [18], 5% (8) DARPA's CGC binaries (CGC) [20], 4% (6) rely on Magma [21], and 1% (2) build on Unibench [22] for benchmarking purposes. Despite its success, LAVA-M is nowadays considered flawed because it artificially injects vulnerabilities into a given target program that are easy for a fuzzer to find but do not correspond to real bugs [17, 24, 25, 26]. More recent works using LAVA-M often do so only for comparability reasons [31, 40]. Similar to LAVA-M, CGC is widely considered outdated and inadequate.

Real-world targets are often limited to binary input-affine programs, while benchmarks are not used by the majority of papers. Benchmarks with artificial vulnerabilities are still used.

3.2.3 Evaluation against State of the Art

Comparison with a strong set of existing work helps to demonstrate that a new method is particularly suited to solve a specific problem. Yet, only a few techniques published in the past few years have been broadly incorporated in follow-up work. Instead, the most famous fuzzers extended with new techniques are AFL [174] with 30% (45), AFL++ [175] with 6% (9), libFuzzer [176] with 5% (7), and syzkaller [177] with 4% (6). Interestingly, all of these tools are non-academic works; only for AFL++ a peer-reviewed paper has been published [175]. Contrasting this number, 33% (49) of the proposed tools are not based on any existing tool.

When looking at the fuzzers chosen as baselines for comparison, we find that AFL is compared against by 35% (53) of studies, followed by QSym [173] with 15% (23),

Table 2: Targets fuzzed in five or more analyzed papers (excluding benchmarks). Some papers report generically to evaluate on binutils, while others specify exact targets, such that numbers in practice may differ slightly. © 2024 IEEE

#Uses	Target
25	objdump, readelf
20	nm, tcpdump
19	libpng
17	libtiff
13	cxxfilt, jhead, libjpeg
12	libxml2
11	nasm
10	jasper, libming, openssl, size
9	file, ImageMagick, mjs, tiff2pdf
8	djpeg, exiv2, JavaScriptCore, libarchive, SQLite, v8, xmllint
7	ChakraCore, ffmpeg, harfbuzz
6	binutils, lcms, lrzip, mupdf, OpenJPEG, SpiderMonkey
5	bento, bsdtar, catdoc, cflow, curl, freetype2, GraphicMagick, json, pcre2, proj4, strip, tiff2ps, yara, zlib

AFLFast [178] with 14% (21), Angora [179] with 13% (20), FairFuzz [165] with 8% (12), and AFL++ with 9% (14). From the 150 papers we analyzed, only QSym (2018), FairFuzz (2018), and MOpt [164] (2019) have been chosen by more than five follow-up works for comparison. More recently, only Fuzzilli [41] (published 2023, open-sourced early 2019) was used by multiple works for their evaluation, even before the paper was published. This does not account for techniques replicated in AFL++ or LibAFL [72], which reimplement many successful techniques proposed [153, 164, 178, 180]. On average, a fuzzing paper evaluates against 3.2 other fuzzers.

Analyzing whether papers omit comparing against a relevant fuzzer in their evaluation, we find that 20% (30) of the works ignore at least one relevant state-of-the-art method and 3% (4) even omit comparing against their baseline, i. e., the tool on which they base their own fuzzer.

45% of fuzzing research builds on top of non-academic fuzzers, 33% build a new tool. 23% percent of fuzzing evaluations fail to compare against relevant state-of-the-art fuzzers or their own baseline.

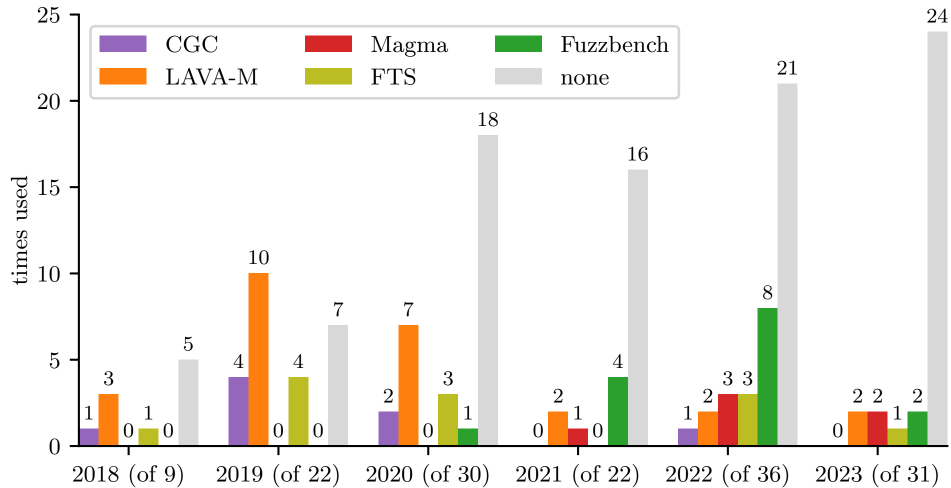


Figure 1: Benchmark usage over the years. The numbers in brackets represent the number of papers analyzed for the respective year. Note that some papers use multiple benchmarks, hence the numbers do not add up. © 2024 IEEE

3.2.4 Evaluation Setup

With respect to the evaluation setup, we analyze the runtime, the number of CPU cores assigned, whether all resources were allocated fairly, and the seeds used for the experiments.

Runtime. Reviewing the experiment setup used across fuzzing evaluations, we find that the majority of papers uses a runtime of 24h, more precisely 56% (84) of the papers run at least one experiment for 24 hours. As Figure 2 outlines, only 27% (40) of the works use a runtime of less than 23 hours, while 29% (44) use an even higher runtime. 5% (8) do not specify their runtime or have no own experiments measuring time.

CPU cores. In terms of CPU cores assigned to fuzzers, we find an inconsistent picture, with a significantly varying number of CPU cores used. The most common result was that 25% (38) of the papers did not specify how many CPU cores they used, 27% (40) used one core, and 8% (12) used two cores.

Fair computing resources. When checking whether the available computing resources were allocated fairly (e. g., the same number of cores were allocated to each

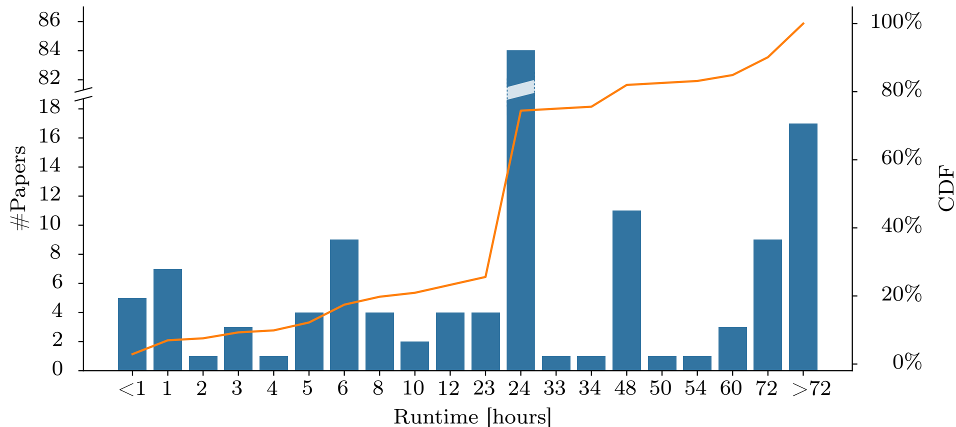


Figure 2: Distribution of runtimes used in practice and cumulative distribution function (CDF), which shows that 27% of papers use a runtime of less than 23 hours. 26 papers use multiple, different runtimes; we include all in these cases. © 2024 IEEE

fuzzer and they were run for the same amount of time), we find that this is the case for 74% (111) of the works. For 15% (23), we could not infer this information from the description in the paper, and 5% (8) did not evaluate other fuzzers or did not conduct any experiments where this was an issue. Crucially, 5% (8) unfairly allocate resources, giving one fuzzer an advantage over another. For these 8, we found one benign case in which an existing method was given more resources, one case in which the number of executions was fairly distributed rather than the runtime (thereby giving slow fuzzers an advantage), two cases in which a different number of cores was used (in one case, giving the new fuzzer twice the cores than others), and four cases where the new approach was allowed some preprocessing time, e. g., for some static analysis pass or seed preprocessing, before it was then allotted the same time as all other tools, effectively giving it more computation time. Unfortunately, the authors rarely explain their motivation for doing so, nor do they consider consequences for the evaluation. Also, our analysis does not address manual work, which may be distributed unfairly between different fuzzers, for example, giving one fuzzer a fine-tuned configuration that performs better.

Initial seeds. Another crucial factor determining a fuzzer's performance is the set of initial seeds [3, 181]. We studied if the *type* of seeds is specified and if information on concrete seed files is available. Out of the 150 papers, 11% (16) require no seeds, 25% (38) use uninformed or empty seeds, 20% (30) use informed seeds, 16% (24) use seeds

provided by the project as test cases or those that are shipped with a benchmark, and 3% (5) use multiple types of seed sets, while 25% (37) do not specify at all what type of seeds are used, making a reproduction challenging. Regarding concrete details, we find that 50% (75) of the papers fail to disclose what seeds they use, compared to 39% (59) that outline their seeds. A further pitfall potentially threatening an evaluation's validity is the fair distribution of the same seeds to all fuzzers. While this is the case in 46% (69) of the studied papers, in 30% (45) of the works this does not become clear, and 5% (8) even use diverging seed sets. Three of these cases arise due to the fuzzer design or other fuzzers lacking the capability to process a particular type of input. We stress that this may be valid, for example, when a fuzzer used for comparison needs a larger seed set than the proposed fuzzer, yet giving a fuzzer a different set of seeds requires special attention and documentation.

We find that 5% of the papers allocate computing resources unfairly, and 5% use different seed sets.

3.2.5 Evaluation Metrics

While many different metrics exist, often specific to the particular technique introduced, a small number of metrics has found widespread adoption: 77% (115) of the papers use some sort of *code coverage*, and 71% (107) use the (re-)discovery of bugs as a metric to compare fuzzers. The third most widespread metric, Time-To-Exposure (TTE), is used by 13% (20) of the papers, mainly from the directed fuzzing domain.

Code Coverage. Code coverage comes in different forms; the most popular are the following: 19% (29) of the papers use branch coverage, 17% (25) employ edge coverage, 13% (19) rely on basic block coverage, and 5% (8) use line coverage on the source code level. Furthermore, 11% (17) use some notion of paths to measure coverage. We stress this metric is unreliable *without* a definition of what the paper considers a path. Differences exist, for example, between actual program paths and AFL's path metric, requiring any paper to specify what they consider a path for their work. Beyond the type of coverage, the process of measuring coverage is also prone to errors, and the concrete choice of measurement is often not documented. In total, we find that 45% (67) of the works lack a clear definition or explanation of how they measure coverage, whereas 32% (48) document this (the remaining papers do not measure coverage). For

example, measuring coverage using a binary with instrumentation that not all fuzzers had access to during the fuzzing campaign gives some fuzzers an advantage. Similarly, when measuring coverage on a bitmap with collisions, the reported coverage is up to 9% smaller [182] than the true one. This may cause problems when a different bitmap size was used during fuzzing, as the inputs saved by a fuzzer may no longer trigger the new coverage on the bitmap with collisions. A further pitfall affects emulation-based fuzzing, especially when using QEMU [183]. We observed that papers often provide no clear distinction between translated blocks as presented by the emulator and actual basic blocks for the target binary. We found that in at least one case this led to overcounting the reached coverage, as translated blocks were mistaken for basic blocks.

Known Bugs. As research from Klees et al. [3] as well as Böhme et al. [15] points out, coverage may not be an accurate proxy for bug finding, even though a strong correlation exists. Ultimately, a fuzzer's goal is finding bugs, making the evaluation of whether it can find known or unknown vulnerabilities an excellent experiment. Known bugs are a good way of measuring a fuzzer's performance, yet it is difficult to find suitable bugs outside well-designed benchmarks, such as Magma [21] or RevBugBench [24].

New Bugs / CVEs. Another commonly used approach is the capability of finding previously unknown bugs. Ethical handling requires researchers to responsibly disclose these bugs to the vendors or maintainers. Both sides can additionally request a CVE that serves as a unique identifier for the found vulnerability. In practice, CVEs have become a commonly used metric to assess whether a fuzzer can find bugs in real-world software, presumably showing its impact. Of the 150 analyzed papers, 59 claim one or more CVEs (9.7 on average, 662 in total). Given the implicit expectation of submissions to have a real-world impact, the authors often try to obtain as many CVEs as possible. We randomly selected 35 of these papers [29, 31, 37, 44, 47, 48, 50, 57, 59, 61, 62, 67, 69, 88, 90, 91, 93, 103, 104, 105, 109, 113, 120, 129, 133, 135, 139, 144, 148, 156, 160, 164, 172, 184, 185] and analyze the 339 CVEs they claim (51% of all CVEs claimed across the 59 papers).

As Figure 3 shows, surprisingly, only 43% (145) of the CVEs are valid (i. e., neither formally disputed, reserved, nor ignored or rejected by the project maintainers) and have been fixed (or at least acknowledged). 26% (88) of the CVEs were still marked as `RESERVED`, preventing us from viewing and analyzing them (all of them were assigned

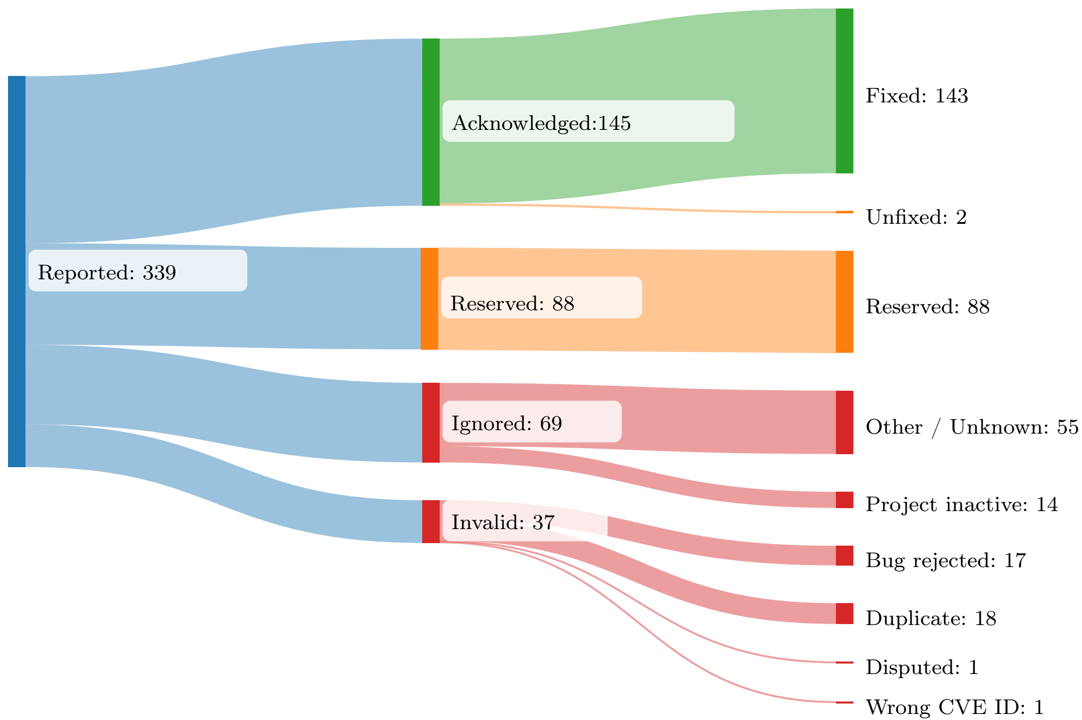


Figure 3: Outcome of 339 CVEs that were reported across 35 papers. Only 43% of the CVEs have been acknowledged by the developers. Pending public disclosure, information on CVEs in the *Reserved* state is withhold. © 2024 IEEE

before 2023). For such CVEs and depending on the assigning authority (called CNA), authors usually have to follow up with the CNA to unblind them once the vulnerabilities are publicly disclosed. Our analysis found 11% (37) of invalid CVEs, including both CVEs that were formally disputed or rejected as duplicates by the assigning CNA, such as Mitre, and such CVEs where the maintainer of the project considered the report to be invalid or not a bug. In one case, the CVE ID specified in the paper did not match the target, leading us to believe the authors mistakenly reported the wrong number. Three CVEs were claimed by more than one paper, raising questions about who identified and reported them initially. A larger number, 20% (69) of the CVEs, have been ignored by the maintainers of the respective projects. Investigating this, we found that in 14 cases, the projects were abandoned several years before the bug was found, or the projects had not found widespread adoption (with a single digit number of stars and forks on GitHub). In these cases, the perceived need to report

many vulnerabilities in a paper appears to be the driving factor in requesting a CVE for such bugs.

Studying why some bug reports were ignored while other bugs were fixed, we found that maintainers tend to ignore issues such as memory leaks in client-side software, for example, an assembler. The reasoning appears to be that the program does not run continuously and is not exposed to external attackers. Many of the ignored CVEs were segmentation faults in `mjs` or `yasm`. The bug tracker of `mjs` appears to be flooded with similar fuzzer-generated bug reports, while the project has not received an update for two years. Similarly, the maintainer of `yasm` has moved to other projects, only occasionally merging pull requests. As security researchers usually only drop the bug details without proposing a patch, these issues remain unfixed. While studying papers, we noticed that several papers claim a specific number of CVEs credited to their work but do not specify any identifier, making it difficult to track them. Interestingly, 18 of the 35 papers report only CVEs that all have been fixed, accounting for 67 of the CVEs.

In summary, the need to show a fuzzer's real-world impact results in a large number of unwarranted CVEs, leading to a situation where only 42% (143) of the 339 assigned CVEs are valid and have been fixed, while many are what one maintainer referred to as "fuzzer fake CVEs" [186]. Creating such invalid vulnerabilities causes multiple problems: It unnecessarily alerts people, reduces maintainer acceptance of fuzzer findings, and raises the expectations for subsequent papers to find a similar number of vulnerabilities.

> 20% of the CVEs have been ignored and remain unfixed, 11% are invalid. 26% are reserved, eluding analysis.

3.2.6 Statistical Evaluation

To confirm the results obtained in the evaluation, a statistical evaluation is highly recommended [3, 187] to detect whether the observed difference is significant or by chance. In practice, the most common approach is to compare the final coverage values achieved by different fuzzers across multiple runs.

In general, a frequently used test for the comparison of the means of two sample sets—such as the coverage values of two fuzzers operating on the same target—is the t-test. While powerful for the detection of differences, it requires strong assumptions.

In particular, the samples have to be approximately normally distributed. This is particularly true for small sample sizes, such as $n \approx 10$. To avoid these strong assumptions, the Mann-Whitney or the similar U-test (called Mann-Whitney U-test to emphasize their equivalence subsequently [188]) is often used. Here, the two samples are assumed to have the same unknown distribution except for a potential shift. The test statistics for the Mann-Whitney U-test is mainly based on the sum of ranks of the two samples in the joint sample. This results in a test for the difference of distribution medians, which is rather robust w.r.t. assumptions that do not hold. For a more detailed discussion of such tests, we refer to Sachs' work [188].

However, the Mann-Whitney U test can have low power, especially for small sample sizes. Suppose, for example, that we have two samples of three runs that achieved the following coverage:

$$x = (1000, 1002, 1001), \quad y = (1208, 1207, 1205)$$

As is easy to see, these samples are strongly separated, and it is hard to explain these results assuming the similarity of the samples' distributions. Yet, the Mann-Whitney U test will not reject the hypothesis of no difference for a significance level $\alpha = 5\%$. Even worse, it will never reject samples of this size, since it only uses the ordering of the observations, and the probability of two samples of size 3 generated from the same distribution to show this pattern of full separation on the real line has a probability $> 5\%$. In other words, we cannot use the Mann-Whitney U test to statistically confirm that the difference between two fuzzers is significant if only three trials have been conducted. Such situations frequently arise if sample sizes are small or observations cannot be approximately described by a parametric distribution that depends only on few parameters, such as a normal distribution.

In summary, a statistical evaluation should use a sufficient number of trials, ideally 10 or more, and use a robust test. Studying the trials used in the 150 analyzed papers, we find that 1, 3, 5, 10, or 20 trials are the most common repetitions chosen. Figure 4 provides a detailed distribution. Overall, 55% (83) of the papers use fewer than 10 trials in at least one experiment (8 papers use a different number of trials throughout their paper). Even worse, 63% (94) conduct no statistical test at all. Only 37% (55) of the papers run a Mann-Whitney U test to measure statistical significance, which—paired with few trials—risks that it may never reject the hypothesis. We find that 15% (22) of the analyzed papers conduct a Mann-Whitney U-test while having five or

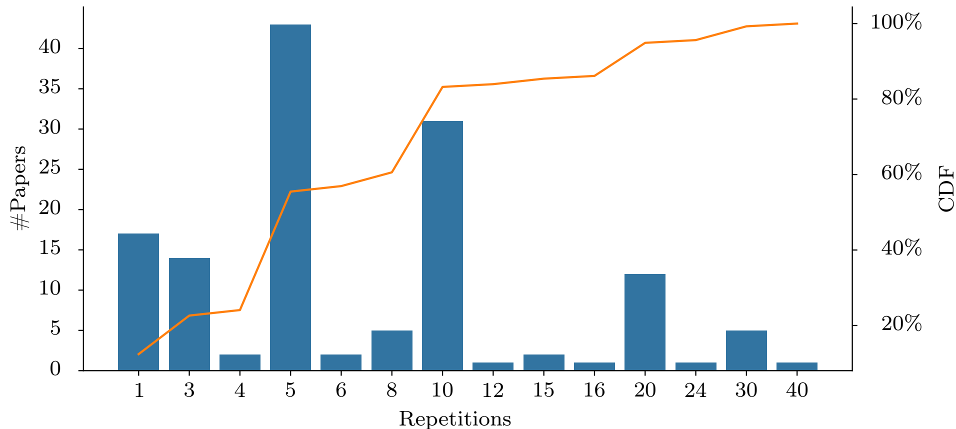


Figure 4: Distribution of trials used in practice and cumulative distribution function (CDF). 8 papers use a different number of trials for different experiments; we include all numbers in this case. Further 21 papers fail to specify the number of trials. © 2024 IEEE

less trials. One work reports p-values without specifying how they have been derived. Interestingly, we found no other tests, such as bootstrap-based ones, being used, despite being recommended by Klees et al. [3]. Beyond measuring statistical significance, it is recommended to quantify the *effect size*, for example, using Vargha and Delaney's $\hat{A}_{12}$ test [189]. Yet, we find that only 10% (15) of studies conduct this test; 2% (3) rely on other means to specify the effect size, leaving us with 88% (132) not using any test to measure the effect size.

Beyond the use of statistical tests, we find that 73% (109) of the papers provide no measure of *uncertainty*, for example, intervals in coverage plots or the standard deviation. This makes it difficult to assess the robustness of reported results, especially considering the inherent randomness in fuzzing runs.

> 63% of the works use no statistical test to assess their results, and 15% use too few trials to achieve robust outcomes. 73% provide no measure of uncertainty.

3.2.7 Threats to Validity

Scientific works often use a dedicated section on *threats to validity* to enumerate, reflect, and address any issue that could potentially render their evaluation invalid. However, when studying how many of the 150 analyzed papers provide such a section, we find that only a minority of 20% (30) of the papers does so.

4 Artifact Evaluation

Beyond studying the evaluation outlined and described in the papers, we select eight papers and study their artifacts. This allows us to assess the practical reproducibility of fuzzing research and provide recommendations grounded in practice. As selection criteria, we pick four recent papers from 2023 and focus on security venues featuring an artifact evaluation. In our experience, papers undergoing an artifact evaluation process provide enhanced documentation and significantly ease the process of setting up a particular tool. However, we test papers that have not undergone artifact evaluation as well to gain a more complete picture. Note that all papers we chose as case studies had attracted our attention during the initial reading for the literature survey in terms of evaluation setup or execution.

In the following, we discuss our lessons learned, pitfalls, and how fuzzing artifacts can be further improved to enhance their reproducibility. Again, we emphasize that it is not our intention to point fingers at specific works but rather to highlight potential pitfalls that researchers in this area should be aware of. More information on all case studies is available in dedicated reproduction repositories on GitHub: `https://github.com/fuzz-evaluator/`. Despite our best efforts, our reproduction may contain errors. If we become aware of any, we will update the respective reproduction repositories on GitHub.

Author Contact. We have anonymously contacted the authors of all case studies and brought up our findings for discussion with them, asking for their help, confirmation, or clarification. Five groups have responded to our mails. Where desired by the authors, we publish a statement of them alongside our reproduction artifact.

Setup. All our experiments were performed on two servers running Ubuntu 22.04 with 196 GB RAM, one with an Intel Xeon Gold 6230R CPU with 52 cores at 2.10GHz, and the other with an Intel Xeon Gold 6230 CPU with 40 cores at 2.10GHz (for consistency, a case study was fully run on one type of server or the other). We use the settings provided by the original papers where sensible, otherwise we run 10 trials for 24 hours each, restricting each fuzzer to a single core.

4.1 Case Study: Artificial Runtime Environment and Unique Crashes

Our first case study is MemLock [120], published at ICSE'20, which proposes to use memory usage as additional feedback. This way, the paper aims to identify resource exhaustion bugs, such as stack exhaustion.

Artifact status. MemLock has undergone artifact evaluation and received the *available* and *reusable* badges. Our additional experiments can be found at `https://github.com/fuzz-evaluator/MemLock-Fuzz-eval`.

Observations. After studying the paper and artifact, we observe the following:

1. According to the artifact but not documented in the paper, the authors artificially alter the runtime environment of one target and lower the maximum stack size. Manually limiting the stack size makes it easier to trigger stack overflow bugs, one of the declared goals of the presented technique.

2. MemLock, similar to many other fuzzing papers, relies on unique crashes as reported by AFL to draw conclusions on the fuzzer's performance. This metric is generally unreliable since a unique crash depends on the set of exercised edges; it does not reflect the number of actual bugs. Here, MemLock's use of the call stack depth as additional feedback may lead to an inflated number of "unique" crashes per root cause.

3. To demonstrate practical impact, MemLock reports 26 CVEs. We found multiple cases among them where up to five CVEs were requested and assigned for a single bug report, to which none of the maintainers responded.

4. MemLock's artifact is based on PerfFuzz [190] (itself an AFL-derivative), but the paper suggests it is based on AFL.

We design three experiments to analyze and reproduce MemLock's performance. For full details, we refer the interested reader to our reproduction artifact.

Experiment 1: Artificial Runtime Limits. We first study the impact of artificially lowering the stack size for the target `flex`, which was not documented in the paper. After recreating the setup and running the fuzzing campaign with and without the artificial limit, we observe that MemLock finds the claimed crashes only with the artificially lowered limit. While memory corruption bugs may warrant discussing artificial scenarios, we believe memory exhaustion created through artificial limits cannot be considered realistic. In any case, we recommend documenting such limits in the paper.

Experiment 2: Unique Crashes. We investigate whether superiority claimed due to *unique crashes* persists when examining the underlying bugs and root causes. Using a developer patch and manual triaging, we identify the underlying bugs for three evaluation targets and find that AFL finds four bugs, while MemLock locates only three, even though it finds significantly more unique crashes.

Experiment 3: Reported CVEs. When studying the reported vulnerabilities, we noticed that six CVEs, CVE-2020-36370 to CVE-2020-36375, refer to the same bug in `mjs`. This bug was never acknowledged by the maintainers of `mjs`. This pattern repeats for other groups of CVEs.

Lessons learned: Unique crashes are not a reliable metric; instead, we suggest using (known) bugs. We recommend not using artificial runtime environments without good reason and, if done, documenting such limits. We strongly recommend against the practice of obtaining as many CVEs as possible. Real-world impact should not be measured based on the number of assigned CVEs.

4.2 Case Study: Exaggerated Vulnerabilities

For the next case study, we selected SoFi [103], published at ACM CCS'21. This work aims to use a reflection-based analysis to create a syntactically and semantically valid but diverse set of seeds for fuzzing JavaScript engines.

Artifact status. Artifact evaluation was not available for SoFi, but the authors released the source code via an independent web page [191]. While trying to set up the artifact, we noticed that crucial parts of the source code were missing. The authors stated they would release the missing pieces once the code is polished [191], but did not

react to our e-mails asking for access to the code. Without a chance to reproduce the artifact, we solely studied the paper, in particular the reported vulnerabilities summarized in Table 2 of their paper, entitled "Summary of discovered vulnerabilities" [103].

Observations. We find that all seven vulnerabilities claimed in the actively used modern browser engines (i. e., v8, SpiderMonkey, and JavaScriptCore) are invalid and have been rejected by the respective developers, six out of seven even *before* the conference submission deadline. While SoFi manages to find confirmed vulnerabilities in other programs, we believe it is important to not oversell results by claiming to have found vulnerabilities in browser engines, when in fact they were not a bug at all. We assume that the bug report IDs were blinded, as is common practice for submission, such that the reviewers could not verify the validity of the presumed vulnerabilities.

Lessons learned: We highly discourage marketing invalid bug reports as a vulnerability. Feedback from the developers must be taken into account (especially if bug reports are rejected by the developers). Pledges to release the source code should be kept.

4.3 Case Study: Missing Baselines

DARWIN [40] was published at NDSS'23 and honored with a *Distinguished Paper Award*. The paper focuses on improving mutation scheduling. More specifically, the authors propose to use an evolution strategy and dynamically adapt the mutation selection to the target under test.

Artifact status. Artifact evaluation was not available to DARWIN, but the authors publicly released an artifact. Our reproduction artifact is available at `https://github.com/fuzz-evaluator/DARWIN-eval`.

Observations. Analyzing the paper and artifact, we found a number of issues:

1. Coverage differences between DARWIN and tested baselines on FuzzBench are not statistically significant nor consistent with the paper's FuzzBench results.

2. The results on MOpt [164] listed in the DARWIN paper indicate that the port implemented for MOpt may have erroneously restricted the number of usable mutations. We find that this strongly influences the results.

3. The artifact appears to be based on Git tag 2.55b of Google's AFL fork and not 2.54b, as listed in the paper.

4. The artifact does not provide the AFL 2.55b port for MOpt or their baseline AFL-S, preventing reproduction or analysis.

We design three experiments to analyze DARWIN. More experiments and details are available in our artifact.

Experiment 1: Coverage. We use FuzzBench to reproduce DARWIN's coverage measurements (in particular, Table III of their paper). Running all targets for 24 hours, we compare it against AFL 2.55b and MOpt, which is based on AFL 2.52b. Notably, we do not use DARWIN as configured in FuzzBench but follow the author's recommended configuration (see Experiment 3). In our FuzzBench results, MOpt does not show the major performance degradation shown in the paper results. Overall, FuzzBench ranks DARWIN above MOpt and AFL, both by score and rank. In individual targets, DARWIN is the best performer in nine of the targets, but only with statistical significance in four. Our results show the difference between DARWIN and its baselines to be less than reported in Table III of their paper. Where they find DARWIN's median relative coverage to be the highest for 15 out of 19 targets, we find this to be the case for 4 out of 18 targets[†] (DARWIN is worse than at least one baseline in two cases and tied with at least one baseline in the other cases). Note that the original paper evaluates over a six hour period instead of the 24 hours recommended by Klees et al. [3]. While we provide the statistical data for the 24 hour data here, we emphasize that the results reported in the paper for the six hour mark are similarly not reproducible and invite the reader to view our full evaluation report data available on GitHub.

In summary, our results show a similar tendency to their paper, but the difference observed between DARWIN and its baselines is smaller. Notably, DARWIN reports a coverage improvement of only 1.73% over AFL, making it difficult to judge the difference between these fuzzers meaningfully.

[†] FuzzBench has meanwhile removed the target `php`.

Table 3: Comparing the code coverage reported by FuzzJIT to our measurements. © 2024 IEEE

	Reported			Measured
Engine	**Fuzzilli**	**FuzzJIT**	**Rel. Increase**	**Rel. Increase**
JSC	16.47%	21.90%	33%	-2%
V8	13.82%	16.67%	21%	-3%
SM	15.53%	17.97%	16%	-12%

Experiment 2: New Baseline. We propose a second baseline to test DARWIN's contribution of a dynamically adapting mutation selection: we replaced its proposed weighting with a random selection (that is reweighted at a constant interval). This implementation, DARWIN$_{RAND}$, provides a new baseline that allows to better judge DARWIN's contribution, as any improvement can be directly attributed to DARWIN's evolutionary algorithm rather than other fuzzer implementation details, such as dynamically adapting mutation selection. We find in our FuzzBench results no statistical significant difference between DARWIN and DARWIN$_{RAND}$, meaning we were unable to demonstrate that the evolutionary aspects of DARWIN's approach significantly contributed to the improvement compared to randomly changing mutation selection over time.

Experiment 3: Per-Seed Mutation Scheduling. After contacting the authors, they noted that the per-seed mutation scheduling (`-p` flag) set by FuzzBench should be disabled for the evaluation because it worsens performance and was not intended as part of the paper. To confirm this, we separately evaluated DARWIN with and without per-seed mutation scheduling on seven targets: we found that disabling the per-seed mutations slightly improved performance overall, leading to higher median coverage in some targets, but not statistically significantly so for any target by Mann-Whitney U. We have used the author-recommended configuration (no `-p` flag) for Experiments 1 and 2.

Lessons learned: A baseline suited to test the proposed technique is necessary to detect differences that can be attributed to the proposed technique rather than the new fuzzer implementation as a whole. We further recommend publishing all evaluation artifacts, also including benchmarking reports and raw data.

Table 4: Comparing the semantic correctness rate reported by FuzzJIT to our measurements.
© 2024 IEEE

	FuzzJIT		Fuzzilli	
Engine	**Reported**	**Measured**	**Reported**	**Measured**
JSC	90.33%	65.88%	62.80%	66.56%
V8	97.04%	63.67%	64.34%	66.74%
SM	93.28%	63.93%	64.13%	67.47%

4.4 Case Study: Non-reproducible Measurements

A recent paper published at USENIX'23, FuzzJIT [55], aims to detect bugs in JIT compilers, including those used in modern browsers.

Artifact Status. FuzzJIT underwent artifact evaluation and was awarded the *available* and *functional* badges. Our reproduction artifact can be found at: `https://github.com/fuzz-evaluator/fuzzjit-eval`.

Observations. After studying the paper and testing the artifact, we observe several shortcomings:

1. Coverage does not reproduce as outlined in the paper; in our experiments, FuzzJIT performed worse than Fuzzilli on all targets.
2. Reported improvements of the semantic correctness rate did not materialize in our experiments.
3. It is not possible to study the bugs found because the time frame, engine versions, and resources spent were not specified in the paper, hindering fair reproduction.

We design two experiments to analyze the claims of FuzzJIT in more detail.

Experiment 1: Code Coverage. When trying to reproduce code coverage, we find significantly different results. As shown in Table 3, FuzzJIT reports a code coverage improvement of up to 33% over Fuzzilli. In stark contrast, our experiments show a code coverage decrease of -2% to -12%. Despite searching for the cause, we find none

explaining this difference. We speculate that the negative outcome of the comparison experiment is a consequence of benchmarking with different versions of Fuzzilli. This is based on the observation that the state-of-the-art fuzzers compared to in the evaluation are taken from UniFuzz [22], which uses an outdated version of Fuzzilli; FuzzJIT itself is based on a more recent version of Fuzzilli. Unfortunately, the authors have not responded to our request for help.

Experiment 2: Semantic Correctness Rate. Besides code coverage, FuzzJIT also evaluates the semantic correctness rate of generated samples, i. e., the number of samples that do not raise an uncaught exception during execution in the JS engine. As shown in Table 4, we could *not* measure any improvement of the semantic correctness rate, contrasting the paper's claim of a significant improvement.

> **Lessons learned:** Relying on outdated baseline versions can create a distorted picture of a fuzzer's performance. Authors should ensure that they use the latest version of all tools for comparison.

4.5 Case Study: Uncommon Metrics

Published at USENIX'20, EcoFuzz [133] proposes to replace AFL's seed scheduling algorithm with a version relying on the adversarial multi-armed bandit model. This way, EcoFuzz finds more paths while generating less seeds.

Artifact status. EcoFuzz has undergone artifact evaluation and was awarded the *passed* badge, indicating that the artifact is available and ready to be reproduced. Our independent reproduction repository is located online at `https://github.com/fuzz-evaluator/EcoFuzz-eval`.

Observations. When studying the paper and artifact, we noticed that the evaluation deviates from typical fuzzing evaluations: The work does not report achieved code coverage over time. Instead, the paper visualizes the total number of paths discovered over executions. This aligns with the paper's goal of finding more path (bandits in EcoFuzz's multi-armed bandit model) with fewer executions (trials in the model). The presented results may lead readers to infer that a higher number of total paths equates to higher code coverage, which is not necessarily true.

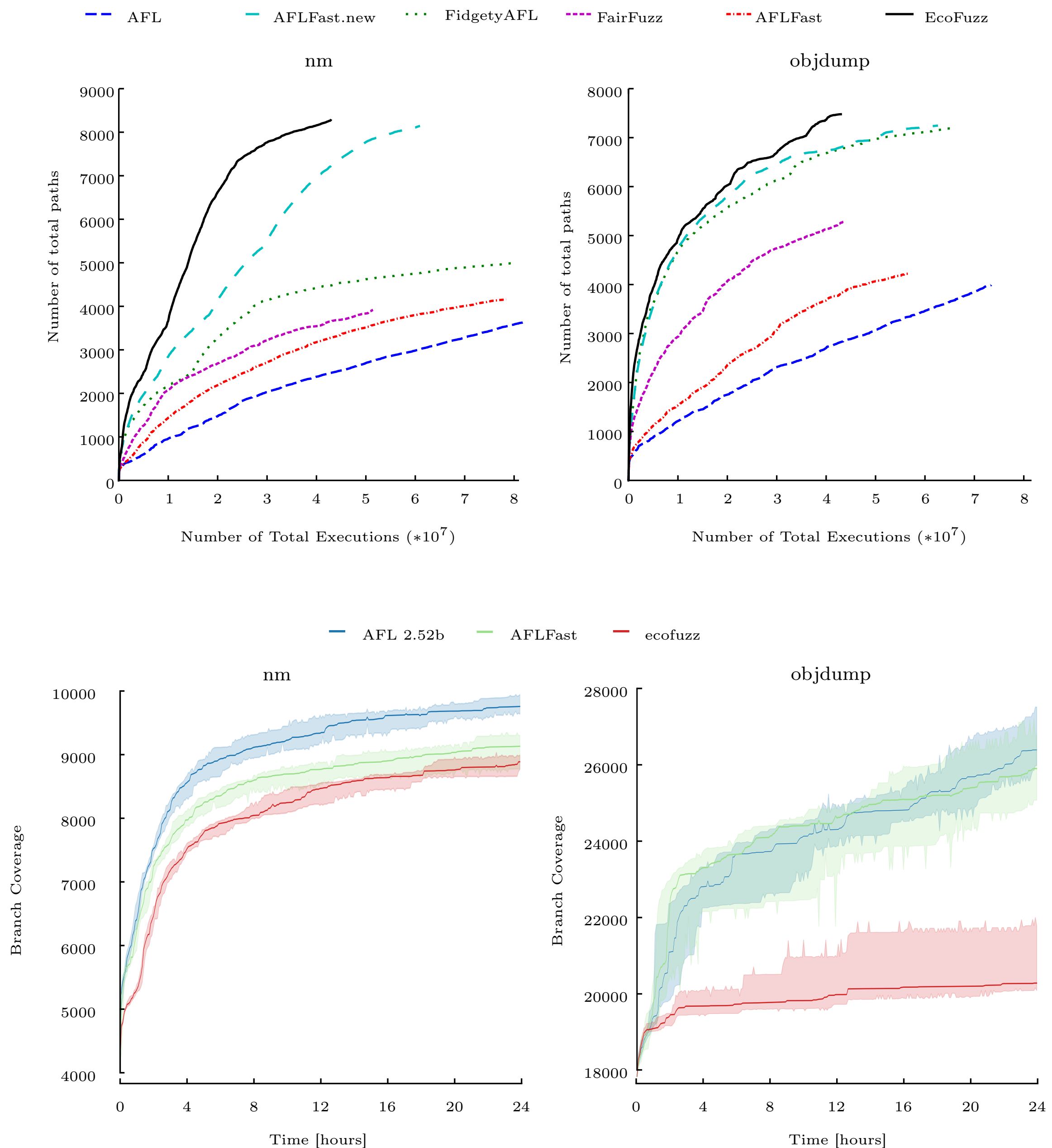


Figure 5: The upper two graphs published in the EcoFuzz paper [133] show a strong advantage over all competitors on the non-standard metric *number of totals paths over the number of total executions*. The two plots at the bottom compare EcoFuzz on the standard metric *branch coverage over time*. On the standard metric, EcoFuzz performs significantly worse. © 2024 IEEE

Experiment: Code Coverage. We design an experiment in FuzzBench where we compare EcoFuzz against its best-performing competitor, AFLFast, and its baseline, AFL. We test these fuzzers on three targets, `nm`, `libpng`, and `objdump`, where the original evaluation[‡] found EcoFuzz to be the fuzzer to find the *most* paths. Our results, shown in Figure 5, demonstrate that EcoFuzz achieves *less* code coverage than the other fuzzers in all scenarios, except for a statistically insignificant one, where it performs similar to AFLFast on `libpng`. This underlines that finding more paths does not necessarily translate to achieving a higher coverage. The full results and the generated FuzzBench reports can be found in our reproduction repository.

Corresponding with the authors, they state they have been following fuzzing evaluations at the time that focused on path coverage, and they have confirmed that EcoFuzz may cover fewer branches on some binaries, stating that its goal is to optimize for paths over executions rather than branches over time.

Lessons learned: A fuzzer may excel at one metric but not on another; hence, selecting a suitable set of evaluation metrics is crucial to provide a reader with the full picture. Evaluating on established metrics is required, as new metrics may imply a completely different picture.

4.6 Case Study: Unclear Documentation

Another paper published at USENIX'23, Polyfuzz [57], targets programs containing code in different languages, such as interpreter languages calling into native bindings.

Artifact status. PolyFuzz has been awarded the *available* badge. Our reproduction artifact is available at `https://github.com/fuzz-evaluator/PolyFuzz-eval`.

Observations. While studying the artifact, we noticed irregularities regarding the seed sets used by PolyFuzz compared to the other fuzzers. An example of such a case is the `image_load` harness for the Python image processing library Pillow. In this particular case, the fuzzer Atheris gets 39 seed files, while PolyFuzz's seed directory has 58 files.

[‡] The evaluation used `readpng`, which internally uses `libpng`, while we use `libpng_read_fuzzer` as bundled with FuzzBench.

Experiment: Fair seed allocation. We intended to run both fuzzers with their respective seed sets to measure the impact of these different seed sets on the coverage. Unfortunately, the authors' extension of Atheris (called Atheris-Cext in the PolyFuzz paper), which would allow to compute combined coverage for both Python and the native code, was not released alongside their artifact. Hence, as proxy measurement, we compute the initial coverage achieved by PolyFuzz on both seed sets. For the seed set given to Atheris, PolyFuzz covers 218 edges, while for its own seed set, it covers 814 edges. Evidently, one seed set provides more than three times as much coverage as the other, giving PolyFuzz a headstart during the evaluation.

When contacted, the authors clarified that they did not keep the seed sets from their evaluation, but they assured us that they used the seeds from the corresponding benchmarks for all fuzzers.

Lessons learned: Seeds have an impact on fuzzer performance. We recommend to give all fuzzers the same set of seeds and to publish the seeds used.

4.7 Case Study: Incomplete Artifact

Firm-AFL [161], published at USENIX Security'19, aims to fuzz Linux-based IoT firmware via augmented process emulation. To do so, the core fuzzing loop targets a single binary under user-mode emulation, while selectively forwarding system calls to a full-system emulator.

Artifact status. Artifact evaluation was not available to Firm-AFL, but different versions of its source code are publicly available across multiple repositories. Our reproduction artifact is available at `https://github.com/fuzz-evaluator/firmafl-eval/`.

Observations. During our analysis of the artifact, we noticed that the repository lacks documentation. Crucial steps are missing, like correct build instructions for different configurations, making it hard for researchers to reuse the artifact and set up the fuzzer and its environment correctly. Furthermore, when setting up the experiments, we noticed that some of the experiment configuration files were missing and target harnessing is heavily inlined with core emulation logic. Not only do these issues hinder extensibility, but they also prevented us from getting all targets working to reproduce

the Firm-AFL experiments. The fuzzer binaries are shipped in a pre-compiled binary version and fail to build from the provided source code. Moreover, the provided baseline uses an older version of AFL (2.06b), while the augmented mode uses AFL v2.52b.

Experiment: Crash Triggers. Being the only experiment with enough documentation to reproduce, we aim to measure the number of crashes produced by both the augmented and full-system emulator versions. We were able to run fuzzing campaigns for 9 out of 11 targets, where one of them only ran for the baseline and not Firm-AFL. The remaining two targets lack the required target-specific configurations. Unfortunately, we could only partially reproduce the claims as presented in the Firm-AFL paper and observed one case where the baseline performed better than Firm-AFL. The full results of our experiments can be found in our reproduction repository.

Lessons learned: While it is unreasonable to expect each academic artifact to be of production quality, we recommend to strive for a reasonable level of readability and documentation that allows others to understand and use the code, thus promoting reproducibility.

4.8 Case Study: Unfair Coverage Measurements

The final case study analyzes FishFuzz [48], published at USENIX'23. The paper proposes an input prioritization strategy based on a multi-distance metric that allows for optimizing the fuzzing efforts towards thousands of targets (e. g., sanitizer labels) in the sense of direct fuzzing.

Artifact status. FishFuzz has received the *available* and *functional* badges. Our additional experiments are available at `https://github.com/fuzz-evaluator/FishFuzz-eval`.

Observations. When studying the artifact in detail, we notice that FishFuzz's way of measuring coverage may erroneously give FishFuzz an unfair edge. From all evaluated fuzzers, FishFuzz was the only fuzzer to place coverage instrumentation not only within the actual target but also in the added ASAN instrumentation. Consequently, FishFuzz also stored inputs that exercised new coverage in the instrumentation; other fuzzers discarded these inputs, as no new coverage was observed. This became a problem

when the binary instrumented by FishFuzz was used for coverage measurements for all fuzzers during evaluation since—by design—only FishFuzz would keep inputs exercising coverage in the ASAN instrumentation.

Experiment: Fair coverage measurement. To demonstrate the impact of measuring coverage in instrumentation code, we measure the coverage for a binary both with and without FishFuzz instrumentation. The result is depicted in Figure 6. If the FishFuzz coverage binary is used for coverage computation, FishFuzz covers 8.44% more edges on average over all runs. When using a binary with standard AFL instrumentation (i. e., where coverage is not measured in the additional instrumentation), the observed coverage increase is reduced to 1.69%. Furthermore, the total number of edges is considerably smaller, showing that edge counts between different binaries do not translate. Note that both coverage binaries rely on colliding bitmaps since the artifact tooling of FishFuzz expects standard AFL bitmaps to be used. We recommend to not use colliding bitmaps for coverage measurements.

Lessons learned: Unintended side effects may skew coverage measurements; we recommend using standardized methods of measuring coverage.

5 Revised Best Practices for Evaluation

Based on our literature analysis and the case studies, we now provide recommendations on ensuring a fair and reproducible fuzzing evaluation. A comprehensive checklist that summarizes these recommendations is available in our GitHub repository at `https://github.com/fuzz-evaluator/guidelines`. Overall, we recommend that authors thoroughly review the *threats to validity* for their respective works to reflect potential issues that could invalidate their evaluation.

5.1 Reproducible Artifact

For reproducibility, it is crucial to open-source the source code including documentation. We highly recommend participating in an artifact evaluation if available. Furthermore, it is essential to (i) specify the exact versions of targets (and harnesses) and fuzzers used for comparison, (ii) use runtime environment abstractions, such as Docker (where feasible), (iii) name the baseline on which the new technique is implemented

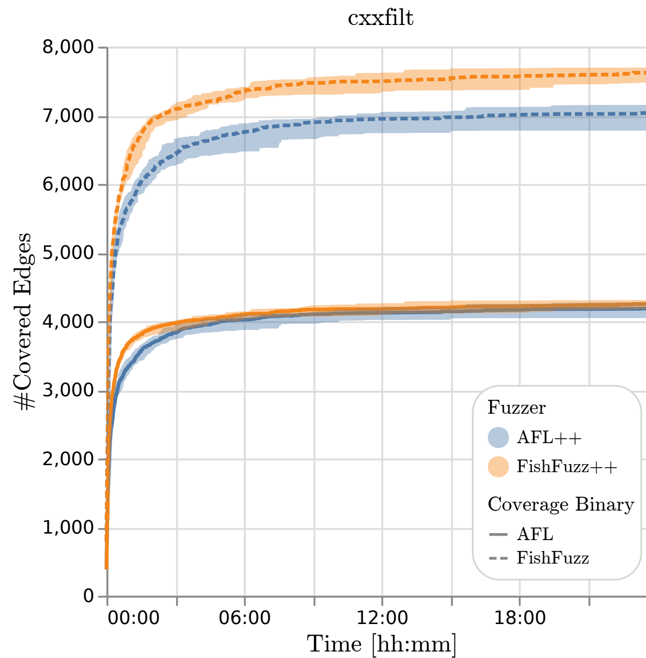


Figure 6: Median coverage over time for `cxxfilt`: In one case, we measure coverage via a standard AFL binary and, in the other we use FishFuzz's binary that contains additional coverage instrumentation. For each fuzzer, the target was run 10 times for 24h each. The displayed intervals enclose all ten runs of the respective fuzzer. If the coverage is measured on the biased binary with FishFuzz instrumentation (- -), FishFuzz++ finds on average 8.44% more edges than AFL++. Measuring coverage on a standard AFL binary (—) (without additional instrumentation introduced by FishFuzz), the coverage delta is only 1.69%. © 2024 IEEE

upon (if any) as well as its version, and avoid squashing commits of this baseline. In the long term, a mandatory artifact evaluation as part of the submission process could improve the quality and reproducibility of research artifacts.

5.2 Targets under Test

Selected evaluation targets should form a representative set that shows strengths of the proposed approach and allows for comparability with previous work. It is therefore desirable to include targets that have been tested in other works. Actions such as patches applied to targets should be explained. If a fuzzer has certain restrictions (such as symbolic execution-based techniques not being able of modeling all syscalls), we recommend

outlining those. We also highly recommend using well-established benchmarks, such as FuzzBench, to facilitate easy reproducibility.

5.3 Comparison to Other Fuzzers

It is crucial to compare against the state of the art in the respective field and the baseline (if any) on which the new technique is implemented. This also includes well-established and actively maintained fuzzers, such as AFL++. Including the new fuzzer in benchmarks such as FuzzBench allows for comparing against a wide range of fuzzers. If presenting a new technique with separable design choices, review them individually via ablation studies, for example, by designing baselines that successively enable or disable individual components.

5.4 Evaluation Setup

The chosen evaluation setup should be well documented. This entails details regarding the used hardware, experiment runtime, number of allocated cores, and processes per fuzzer. The conducted experiments and how to reproduce them should be explained.

For the runtime, we recommend choosing at least 24 hours. Longer runtimes may be appropriate if the evaluated fuzzers do not flatline at the end of the experiment. Regarding CPU cores, choosing a single core may not be representative of modern systems. Special care must be taken to avoid congestion in the kernel when running multiple fuzzers in parallel on one system; even if using Docker, the kernel may become a bottleneck in resolving certain syscalls, unfairly slowing down one fuzzing process. Individual fuzzer instances can be encapsulated in separate virtual machine instances to avoid such situations.

Regarding seeds, we recommend running with uninformed seeds or multiple seed sets. Seeds must be described and accessible (in the case of informed seeds) to allow for reproducibility. All fuzzers should have fair access to all seeds. If using informed seeds, we recommend plotting or analyzing the coverage achieved by the initial seed set. This avoids attributing a high coverage achieved to fuzzer performance instead of the initial seeds.

5.5 Evaluation Metrics

A fuzzer comparison should use standardized, well-established metrics (at least as a complementary metric if a technique requires the introduction of a new metric); this includes both coverage and found bugs. Optimally, both code coverage and bug-finding capability are evaluated, as both suffer from individual drawbacks [3, 15, 192]. We recommend using modern benchmarks that aid in setting up the experiment and ensure a fair, bias-free execution.

It is necessary to specify details such as how coverage is collected, for example, whether it is measured on a non-instrumented binary, translated blocks from an emulator, or using established means such as `lcov`. Ideally, coverage is not measured using bitmaps with collisions, but using a collision-free encoding or other means. Additionally, the evaluation must ensure that the same notion of coverage is used for each of the compared fuzzers.

When searching for bugs in new targets to show real-world impact, it is crucial to select reasonable targets, i. e., projects that are not insecure by design, have been inactive for years, or are unsuitable for other reasons. We also recommend running other state-of-the-art fuzzers to see whether they find the bugs as well, thereby addressing concerns regarding fuzzing previously untested software. Crashes identified by the fuzzer should be deduplicated before opening a report, and the triaging process should be clearly described. When testing crashes, we recommend reproducing them on a binary without fuzzer or coverage instrumentation to avoid reproducibility issues.

Ideally, only maintainers should request CVEs. If they do not request one, researchers can still link to the bug report instead. Requesting multiple CVEs for a single bug or requesting CVEs without coordinating or informing the maintainers must be avoided. If possible, reporting bugs or CVEs anonymously allows for providing the reviewers with access during submission, such that they can inspect the CVEs or bug reports and assess their validity (as opposed to the current practice of blinding CVEs and bug reports during submission, preventing any analysis by reviewers). That said, we do not believe that having CVEs should be required to show the practical impact of a fuzzer.

5.6 Statistical Evaluation

Any evaluation should be backed by statistical tests. To enable these tests, we recommend running at least ten trials. Alternatively, the number of trials can be calculated via an a-priori power analysis to ensure a sufficient sample size leading to statistically significant results [193]. This is particularly important if the fuzzer under consideration only slightly outperforms the state of the art, where $n \gg 10$ may be required. To avoid the problems mentioned in Section 3.2.6, we recommend an alternative to the widely used Mann-Whitney-U test; permutation tests or resampling tests such as bootstrap methods. These methods avoid strong assumptions regarding a normal distribution.

If more than two fuzzers have been compared for a target, the (bootstrap-based) two-sample t-test is not a good choice, since we would have to perform more than one pairwise comparison to test the null hypotheses of no difference between any of the expected means for the fuzzing methods. This results in the *multiple testing problem*, which is the observation that the probability of at least one false positive result in the set of comparisons performed for a target exceeds the single test level α substantially. The same argument holds for other strategies based on two-sample comparisons such as the Mann-Whitney-U test [194].

A solution to this problem is the bootstrap version of the *ANOVA method*. If the ANOVA rejects the null hypothesis, it shows at level α that there is at least one pair of fuzzing methods that perform significantly different for the target considered. In a second step, a so-called *Posthoc*-test is performed to determine which pairwise comparisons are significant, *given that the ANOVA has already shown that there are significant differences at all.* Possible *Posthoc*-tests are, for example, the Tukey-Kramer method if all pairwise comparisons among all samples are of interest or the Dunnett method if only the comparisons to a reference method, such as the newly developed fuzzer, are of interest [188]. For a bootstrap version of these algorithms, we propose as a simple solution two-sample t-tests with critical values for rejection based on a bootstrap resampling with replacement of the test statistics. Here, for each simulation, the maximum value of the test statistics is used for all pairwise comparisons of interest. We provide more details, algorithms, and scripts implementing examples for these tests in our artifact at `https://github.com/fuzz-evaluator/statistics`. Additionally, evaluations should measure *effect size*, e. g., using Vargha and Delaney's $\hat{A}_{12}$ test [189], and quantify *uncertainty*, for example, by using intervals in plots.

6 Conclusion

Reproducibility is a cornerstone of science and the basis for research. In this work, we have systematically studied how 150 fuzzing papers published in the past six years at leading conferences design their evaluation. Furthermore, we have performed an in-depth analysis of the artifacts of eight papers and attempted to reproduce their results. Based on the insights gained, we outlined several potential pitfalls and shortcomings threatening the validity of fuzzing evaluations. Ultimately, we provided revised recommendations and best practices to improve future evaluation of fuzzing research. We published a concise set of guidelines at `https://github.com/fuzz-evaluator/guidelines` and welcome community contributions.

Acknowledgment

We thank our anonymous shepherd and reviewers for their valuable feedback. Further, we thank Dominik Maier, Johannes Willbold, Daniel Klischies, Merlin Chlosta, and Marcel Böhme (in no particular order) for their helpful comments on a draft of this work. We also thank the countless researchers with whom we have discussed fuzzing research and how to evaluate it, ultimately paving the way for this work. This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and the German Federal Ministry of Education and Research under the grants KMU-Fuzz (16KIS1898) and CPSec (16KIS1899). Additionally, this research was partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant EP/V000454/1. The results feed into DsbDtech.

References

[1] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software. `https://github.com/google/oss-fuzz`.

[2] Association for Computing Machinery. Artifact Review and Badging Version 1.1. `https://www.acm.org/publications/policies/artifact-review-and-badging-current`, 2020.

[3] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[4] Martin Abadi and Roger Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[5] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. SoK: Benchmarking Flaws in Systems Security. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[6] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. Reproducibility and Replicability of Web Measurement Studies. In *ACM Web Conference 2022*, 2022.

[7] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *USENIX Security Symposium*, 2022.

[8] Lin Padgham, Young Lee, Shazia Sadiq, Michael Winikoff, Alan Fekete, Stephen MacDonell, Dali Kaafar, and Stefanie Zollmann. CORE Rankings. `https://www.core.edu.au/conference-portal`.

[9] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The Fuzzing Book. `https://www.fuzzingbook.org/`, 2019.

[10] Valentin J. M. 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*, 47(11):2312–2331, 2021.

[11] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[12] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and Reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[13] Marcel Böhme and Brandon Falk. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[14] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. Estimating Residual Risk in Greybox Fuzzing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[15] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[16] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. Reachable Coverage: Estimating Saturation in Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2023.

[17] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[18] Google. Fuzzer-Test-Suite. `https://github.com/google/fuzzer-test-suite`, 2016.

[19] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[20] DARPA. DARPA Cyber Grand Challenge. `https://github.com/CyberGrandChallenge`, 2018.

[21] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A Ground-Truth Fuzzing Benchmark. *ACM on Measurement and Analysis of Computing Systems (POMACS)*, 4(3):49:1–49:29, 2020.

[22] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for

Evaluating Fuzzers. In *USENIX Security Symposium*, 2021.

[23] Roberto Natella and Van-Thuan Pham. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[24] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *USENIX Security Symposium*, 2022.

[25] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. Evaluating Synthetic Bugs. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

[26] 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. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[27] Haoyu Wang, Junjie Chen, Chuyue Xie, Shuang Liu, Zan Wang, Qingchao Shen, and Yingquan Zhao. MLIRSmith: Random Program Generation for Fuzzing MLIR Compiler Infrastructure. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[28] Ahmad Humayun, Yaoxuan Wu, Miryung Kim, and Muhammad Ali Gulzar. NaturalFuzz: Natural Input Generation for Big Data Analytics. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[29] Yuwei Liu, Siqi Chen, Yuchong Xie, Yanhao Wang, Libo Chen, Bin Wang, Yingming Zeng, Zhi Xue, and Purui Su. VD-Guard: DMA Guided Fuzzing for Hypervisor Virtual Device. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[30] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. SJFuzz: Seed & Mutator Scheduling for JVM Fuzzing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.

[31] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. Evaluating and Improving Hybrid Fuzzing. In *IEEE/ACM International Con-*

ference on Automated Software Engineering (ASE), 2023.

[32] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[33] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. JITfuzz: Coverage-Guided Fuzzing for JVM Just-in-Time Compilers. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[34] Suyue Guo, Xinyu Wan, Wei You, Bin Liang, Wenchang Shi, Yiwei Zhang, Jianjun Huang, and Jian Zhang. Operand-Variation-Oriented Differential Analysis for Fuzzing Binding Calls in PDF Readers. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[35] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[36] Yunhang Zhang, Chengbin Pang, Stefan Nagy, Xun Chen, and Jun Xu. Profile-guided System Optimizations for Accelerated Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[37] Peng Deng, Zhemin Yang, Lei Zhang, Guangliang Yang, Wenzheng Hong, Yuan Zhang, and Min Yang. NestFuzz: Enhancing Fuzzing with Comprehensive Understanding of Input Processing Logic. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[38] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. HOPPER: Interpretative Fuzzing for Libraries. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[39] Ruijie Meng, George Pirlea, Abhik Roychoudhury, and Ilya Sergey. Greybox Fuzzing of Distributed Systems. In *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[40] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: Survival of the Fittest Fuzzing Mutators. In

Symposium on Network and Distributed System Security (NDSS), 2023.

[41] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[42] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[43] Changhua Luo, Wei Meng, and Penghui Li. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[44] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. TEEzz: Fuzzing Trusted Applications on COTS Android Devices. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[45] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. VIDEZZO: Dependency-aware Virtual Device Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[46] Ji Shi, Zhun Wang, Zhiyao Feng, Yang Lan, Shisong Qin, Wei You, Wei Zou, Mathias Payer, and Chao Zhang. AIFORE: Smart Fuzzing Based on Automatic Input Format Reverse Engineering. In *USENIX Security Symposium*, 2023.

[47] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations. In *USENIX Security Symposium*, 2023.

[48] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *USENIX Security Symposium*, 2023.

[49] Lukas Seidel, Dominik Maier, and Marius Muench. Forming Faster Firmware Fuzzers. In *USENIX Security Symposium*, 2023.

[50] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P. Kemerlis. IvySyn: Automated Vulnerability Discovery in Deep Learning Frame-

works. In *USENIX Security Symposium*, 2023.

[51] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In *USENIX Security Symposium*, 2023.

[52] Leo Stone, Rishi Ranjan, Stefan Nagy, and Matthew Hicks. No Linux, No Problem: Fast and Correct Windows Binary Fuzzing via Target-embedded Snapshotting. In *USENIX Security Symposium*, 2023.

[53] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. FirmSolo: Enabling Dynamic Analysis of Binary Linux-based IoT Kernel Modules. In *USENIX Security Symposium*, 2023.

[54] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *USENIX Security Symposium*, 2023.

[55] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *USENIX Security Symposium*, 2023.

[56] Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave (Jing) Tian, and Mathias Payer. GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation. In *USENIX Security Symposium*, 2023.

[57] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *USENIX Security Symposium*, 2023.

[58] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. Griffin: Grammar-Free DBMS Fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[59] Yuanping Yu, Xiangkun Jia, Yuwei Liu, Yanhao Wang, Qian Sang, Chao Zhang, and Purui Su. HTFuzz: Heap Operation Sequence Sensitive Fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[60] Taotao Gu, Xiang Li, Shuaibing Lu, Jianwen Tian, Yuanping Nie, Xiaohui Kuang, Zhechao Lin, Chenyifan Liu, Jie Liang, and Yu Jiang. Group-based Corpus Scheduling for Parallel Fuzzing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[61] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. Minerva: Browser API Fuzzing with Dynamic mod-ref Analysis. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[62] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. μAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[63] Hoang Lam Nguyen and Lars Grunske. BEDIVFUZZ: Integrating Behavioral Diversity into Generator-based Fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[64] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[65] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[66] Harrison Green and Thanassis Avgerinos. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[67] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. Linear-time Temporal Logic guided Greybox Fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[68] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing. In *IEEE/ACM International Conference on Automated Software*

Engineering (ASE), 2022.

[69] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[70] Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. Evocatio: Conjuring Bug Capabilities from a Single PoC. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[71] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[72] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[73] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. MC2: Rigorous and Efficient Directed Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[74] Zitai Chen, Sam L. Thomas, and Flavio D. Garcia. MetaEmu: An Architecture Agnostic Rehosting Framework for Automotive Firmware. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[75] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, and Zhi Xue. SFuzz: Slice-based Fuzzing for Real-Time Operating Systems. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[76] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[77] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. In *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[78] Peng Xu, Yanhao Wang, Hong Hu, and Purui Su. COOPER: Testing the Binding Code of Scripting Languages with Cooperative Mutation. In *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[79] Gen Zhang, Pengfei Wang, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, and Kai Lu. MobFuzz: Adaptive Multi-objective Optimization in Gray-box Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[80] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[81] Dongdong She, Abhishek Shah, and Suman Jana. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[82] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[83] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[84] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. PATA: Fuzzing with Path Aware Taint Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[85] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds. In *USENIX Security Symposium*, 2022.

[86] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths. In *USENIX Security Symposium*, 2022.

[87] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *USENIX Security Symposium*, 2022.

[88] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference. In *USENIX Security Symposium*, 2022.

[89] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing. In *USENIX Security Symposium*, 2022.

[90] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *USENIX Security Symposium*, 2022.

[91] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful Greybox Fuzzing. In *USENIX Security Symposium*, 2022.

[92] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *USENIX Security Symposium*, 2022.

[93] Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. InstruGuard: Find and Fix Instrumentation Errors for Coverage-based Greybox Fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[94] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[95] Qian Zhang, Jiyuan Wang, and Miryung Kim. HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence for Heterogeneous Applications. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[96] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing Symbolic Expressions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[97] Vasudev Vikram, Rohan Padhye, and Koushik Sen. Growing A Test Corpus with Bonsai Fuzzing. In *IEEE/ACM International Conference on Automated Software*

Engineering (ASE), 2021.

[98] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. JEST: N+1 -version Differential Testing of Both JavaScript Engines and Specification. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[99] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[100] Xiaogang Zhu and Marcel Böhme. Regression Greybox Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[101] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[102] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[103] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[104] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[105] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[106] Sebastian Poeplau and Aurélien Francillon. SymQEMU: Compilation-based Symbolic Execution for Binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[107] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[108] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[109] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[110] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided Directed Greybox Fuzzing. In *USENIX Security Symposium*, 2021.

[111] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[112] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The Use of Likely Invariants as Feedback for Fuzzers. In *USENIX Security Symposium*, 2021.

[113] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-Level Fuzzing. In *USENIX Security Symposium*, 2021.

[114] Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. MoFuzz: A Fuzzer Suite for Testing Model-Driven Software Engineering Tools. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[115] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[116] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[117] Suhwan Song, Chengyu Song, Yeongjin Jang, and Byoungyoung Lee. CrFuzz: Fuzzing Multi-purpose Programs through Input Validation. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[118] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. MTFuzz: Fuzzing with a Multi-task Neural Network. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[119] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding Greybox Fuzzing towards Combinatorial Difference. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[120] 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 *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[121] Valentin Wüstholz and Maria Christakis. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[122] 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 *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[123] Wen Xu, Soyeon Park, and Taesoo Kim. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[124] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Symposium on*

Network and Distributed System Security (NDSS), 2020.

[125] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[126] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[127] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[128] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[129] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[130] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[131] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. SAVIOR: Towards Bug-Driven Hybrid Testing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[132] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *USENIX Security Symposium*, 2020.

[133] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *USENIX Security Symposium*, 2020.

[134] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *USENIX Security Symposium*, 2020.

[135] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic Fuzzer Generation. In *USENIX Security Symposium*, 2020.

[136] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security Symposium*, 2020.

[137] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *USENIX Security Symposium*, 2020.

[138] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *USENIX Security Symposium*, 2020.

[139] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *USENIX Security Symposium*, 2020.

[140] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *USENIX Security Symposium*, 2020.

[141] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security Symposium*, 2020.

[142] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *USENIX Security Symposium*, 2020.

[143] 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 *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*

(ESEC/FSE), 2019.

[144] Yuting Chen, Ting Su, and Zhendong Su. Deep Differential Testing of JVM Implementations. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[145] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. DifFuzz: Differential Fuzzing for Side-channel Analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[146] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box Concolic Testing on Binary Code. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[147] Wei You, Xuwei Liu, Shiqing Ma, David Mitchel Perry, Xiangyu Zhang, and Bin Liang. SLF: Fuzzing without Valid Seed Inputs. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[148] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware Greybox Fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[149] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[150] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing Deeply Nested Branches. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[151] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[152] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[153] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[154] Stefan Nagy and Matthew Hicks. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[155] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[156] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[157] 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 (S&P)*, 2019.

[158] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[159] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. AntiFuzz: Impeding Fuzzing Audits of Binary Executables. In *USENIX Security Symposium*, 2019.

[160] 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 *USENIX Security Symposium*, 2019.

[161] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, 2019.

[162] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-Fuzzing Techniques. In *USENIX Security Symposium*, 2019.

[163] Tim Blazytko, Cornelius Aschermann, Moritz Schloegel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019.

[164] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, 2019.

[165] Caroline Lemieux and Koushik Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[166] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[167] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[168] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[169] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[170] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[171] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *USENIX Security Symposium*, 2018.

[172] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *USENIX Security Symposium*, 2018.

[173] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[174] Michał Zalewski. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl/`.

[175] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[176] LibFuzzer - A Library for Coverage-guided Wuzz Testing. `https://llvm.org/docs/LibFuzzer.html`.

[177] Dmitry Vyukov and Google. Syzkaller – Kernel Fuzzer. `https://github.com/google/syzkaller`, 2015.

[178] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[179] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[180] lafintel. laf-intel - Circumventing Fuzzing Roadblocks with Compiler Transformations. `https://lafintel.wordpress.com`.

[181] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed Selection for Successful Fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[182] Stephan Lipp, Daniel Elsner, Thomas Hutzelmann, Sebastian Banescu, Alexander Pretschner, and Marcel Böhme. FuzzTastic: A Fine-grained, Fuzzer-agnostic Coverage Analyzer. In *International Conference on Software Engineering (ICSE)*, 2022.

[183] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference (ATC)*, 2005.

[184] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. StochFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[185] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[186] Michael Matz. Comment 1. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=87675#c1`, 2018.

[187] David Paaßen, Sebastian Surminski, Michael Rodler, and Lucas Davi. My Fuzzer Beats Them All! Developing a Framework for Fair Evaluation and Comparison of Fuzzers. In *European Symposium on Research in Computer Security (ESORICS)*, 2021.

[188] Lothar Sachs. *Applied Statistics: A Handbook of Techniques*. Springer Series in Statistics. Springer New York, New York, NY, 2 edition, 1984.

[189] 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.

[190] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically Generating Pathological Inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[191] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. SoFi Artifact. `https://sites.google.com/view/sofi4js/souce-and-data`, 2021.

[192] Andreas Zeller, Sascha Just, and Kai Greshake. When Results Are All That Matters: Consequences. `https://andreas-zeller.blogspot.com/2019/10/when-results-are-all-that-matters.html`, 2019.

[193] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic press, 2013.

[194] Andrea Arcuri and Lionel Briand. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *International Conference on Software Engineering (ICSE)*, 2011.

Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains

Publication Data

Moritz Schloegel, Tim Blazytko, Julius Basler, Fabian Hemmer, and Thorsten Holz. Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains. In *European Symposium on Research in Computer Security (ESORICS)*, 2021.

Permission. *Reproduced with permission from Springer Nature. The final authenticated version is available online at* `https://doi.org/10.1007/978-3-030-88418-5_11`.

Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains

Moritz Schloegel, Tim Blazytko, Julius Basler, Fabian Hemmer, and Thorsten Holz

Ruhr University Bochum

Abstract

In the arms race between binary exploitation techniques and mitigation schemes, *code-reuse attacks* have been proven indispensable. Typically, one of the initial hurdles is that an attacker cannot execute their own code due to countermeasures such as *data execution prevention* (DEP, `W^X`). While this technique is powerful, the task of finding and correctly chaining gadgets remains cumbersome. Although various methods automating this task have been proposed, they either rely on hard-coded heuristics or make specific assumptions about the gadgets' semantics. This not only drastically limits the search space but also sacrifices their capability to find valid chains unless specific gadgets can be located. As a result, they often produce no chain or an incorrect chain that crashes the program. In this paper, we present `SGC`, the first generic approach to identify gadget chains in an automated manner *without* imposing restrictions on the gadgets or limiting its applicability to specific exploitation scenarios. Instead of using heuristics to find a gadget chain, we offload this task to an SMT solver. More specifically, we build a logical formula that encodes the CPU and memory state at the time when the attacker can divert execution flow to the gadget chain, as well as the attacker's desired program state that the gadget chain should construct. In combination with a logical encoding of the data flow between gadgets, we query an SMT solver whether a valid gadget chain exists. If successful, the solver provides a proof of existence in the form of a synthesized gadget chain. This way, we remain fully flexible w.r.t. to the gadgets. In empirical tests, we find that the solver often uses all types of control-flow transfer instructions and even gadgets with side effects. Our evaluation shows that `SGC` successfully finds working gadget chains for real-world exploitation scenarios within minutes, even when all state-of-the-art approaches fail.

1 Introduction

Early exploitation techniques relied on code-injection attacks, where an attacker injects shellcode into the memory space of an application and then executes it. However, quickly established mitigations forced attackers to adapt. Especially the introduction of the `W^X` policy (commonly referred to as *data execution prevention* (DEP)) made the execution of injected code infeasible, as memory is marked as either writable or executable. This forced attackers to develop novel exploitation techniques that *reuse* already existing code (e. g., *return-to-libc*) [1, 2, 3]. As an additional line of defense, modern operating systems randomize a program's address space layout (ASLR). Still, a single *information leak* or small, non-randomized parts of the executable often provide an attacker the capability to mount their attack. In the past years, control-flow integrity (CFI) [4] has gained popularity. This technique enforces the property that only legitimate control-flow transitions inside a benign set required by the program are performed. While greatly limiting the attacker's freedom to chain arbitrary code snippets, so-called *code-reuse attacks* are still feasible in practice [5, 6, 7]. In general, code-reuse attacks have been shown to be Turing complete [8, 9]. Note that in practice, attackers often only need to disable `W^X` before they can execute arbitrary shellcode in the context of the exploited program. This is commonly achieved by chaining so-called *gadgets*, (short) sequences of instructions ending with an indirect control-flow transfer such as `ret` [2]. Even medium-sized programs contain thousands of gadgets, making the process of extracting and finding a suitable combination cumbersome. Various techniques to automate the process were proposed: Initial attempts used pattern-matching-based strategies to identify a chain [10, 11]; later approaches [12, 13, 14] make use of symbolic execution to classify gadgets and identify undesirable side effects, e. g., writing values to memory. However, even the most advanced approaches to date rely on various heuristics to confine the large search space [5, 6, 7]. While sometimes effective, pruning may lead to false negatives: these heuristics try to find generic chains to work across many targets, but in some cases no such chain exists.

In this paper, we propose a novel method to find gadget chains efficiently *without* pruning the search space. One category of tools that particularly excels at finding solutions for decision problems involving a large search space are SMT solvers [15]; they check if a (potentially large) set of logical formulas—so-called *constraints*—can be satisfied [16]. By building a logical formula that describes (1) the CPU and mem-

ory state before executing the first gadget, (2) the CPU and memory state desired by the attacker, and (3) the data flow between gadgets, we can model the gadget chain synthesis as a reachability problem and use an SMT solver to decide it. This approach is similar to bounded model checking [17], a software verification technique used to determine whether a system meets a given set of requirements: it combines a set of assumptions that have to hold before execution *(preconditions)* and a set of requirements that have to hold after execution *(postconditions)* with a logical encoding of the program semantics and then queries an SMT solver. If the solver returns `SAT` (satisfiable), it provides a *model* representing a concrete variable assignment that satisfies the given constraints. In our case, this implies that the solver successfully synthesized a gadget chain. If the result is `UNSAT` (unsatisfiable), the SMT solver mathematically proved that the constraints cannot be satisfied and, thus, no chain can exist for the given set of gadgets.

We introduce the design and implementation of `SGC`, a generic approach capable of automatically identifying gadget chains without relying on any classification or heuristics to prune the search space. At the same time, the logical formula offers a framework to specify target-specific constraints. Our evaluation demonstrates that `SGC` not only outperforms all state-of-the-art tools with regard to finding gadget chains, but the synthesized chains always work in real-world scenarios. For instance, we demonstrate how we can craft a gadget chain that spawns a shell for a stack-based buffer overflow in `dnsmasq`: After defining the concrete CPU state as preconditions, we encode the target state right before executing the system call `execve(&"/bin/sh", 0, 0)`; running `SGC` provides us with a gadget chain spawning the shell without requiring any other information. We further demonstrate that even complex constraints (e. g., the sum of all values in the gadget chain must be equal to a specific value) can be satisfied by `SGC`.

In summary, our main contributions are:

- We introduce a generic approach to synthesize gadget chains in an automated way based on bounded model checking. Our approach does not require heuristics or pruning of the search space; instead, the SMT solver provides a proof of existence in the form of a gadget chain or proves that no gadget chain can be found for the given gadgets and constraints.

- We present the design and evaluation of our prototype SGC. We show that it not only outperforms all state-of-the-art approaches, but also works in real-world settings.
- Our approach provides unprecedented flexibility: SGC allows an attacker to specify arbitrary constraints and, thus, model even complex or unusual exploitation scenarios.

To foster further research in this area, we open-source SGC at https://github.com/RUB-SysSec/gadget_synthesis.

2 Shortcomings of State-of-the-Art Approaches

In the following, we discuss state-of-the-art approaches from academia and industry that can be used in practice to generate gadget chains automatically and analyze their shortcomings in this regard (cf. Table 1). We find that existing tools can be separated into two categories, based on their gadget chain generation:

Hardcoded Chaining Rules. `Ropper` [11] and `ROPgadget` [10] both fall into this category. Their main task is to extract gadgets, but both require hardcoded rules based on regular expressions to chain gadgets. While `ROPgadget` only supports a single exploitation scenario (i. e., building a system call to `execve(&"/bin/sh\0", 0, 0)`), `Ropper` allows system calls to `mprotect` as well. As a result, these tools are inflexible in practice.

Symbolic Exploration. `angrop` [12] and `ROPium` [13] operate on an *intermediate representation* of gadgets, which allows them to symbolically determine side effects and perform a classification. To this end, gadgets are first lifted, then analyzed, and chained together in the last step. The latter usually involves an algorithm such as *depth-first search* (`ROPium`) or *breadth-first search* (`angrop`) to identify a sequence of gadgets that fulfills the attacker's specifications, such as specific argument values. While vastly more flexible than approaches using hardcoded rules, these tools are no panacea. They still rely on a classification of gadgets, and while they provide greater flexibility by allowing simple memory and register constraints, they lack support for more elaborate constraints. `P-SHAPE` by Follner et al. [14] also uses a symbolic exploration approach. However, it only focuses on finding gadgets useful for constructing library calls. It does neither provide a full gadget chain nor allows an attacker to specify any constraints.

Table 1: Features of different tools capable of automatically chaining gadgets.

	SGC	P-SHAPE	angrop	ROPium	ROPgadget	Ropper
supports chains without `ret`	✓	✗	✗	✓	✓	✓
no hardcoded chaining rules	✓	✓	✓	✓	✗	✗
no classification needed	✓	✗	✗	✗	✗	✗
supports arbitrary postconditions	✓	✗	✗	✗	✗	✗

Overall, all approaches lack flexibility; especially, they fail to support arbitrary postconditions (cf. Table 1). Instead, they rely on a classification of gadgets and pre-defined strategies to identify a gadget chain. Even when finding a chain, we empirically observe that they often crash the targeted program, e. g., through invalid memory accesses. Despite this, no tool makes any attempt at verifying the correctness of the generated gadget chains.

3 Design

In the following, we present a gadget-agnostic design that does not perform any preclassification of gadgets while providing high flexibility by allowing to specify arbitrary, complex constraints. The nature of our approach overcomes the limitations of existing approaches. Most importantly, we can enforce an arbitrary CPU register and memory state before and after the exploitation—our design will identify a gadget chain facilitating the transition from the initial to the desired state using any gadgets available, including such using `jmp` and `call` instructions. To this end, our approach encodes the search of the gadget chain as a synthesis problem that an SMT solver decides. More specifically, our design is based on bounded model checking: preconditions and postconditions are represented by the initial and desired CPU state, while a logical formula encodes the possible gadget chain that facilitates the transition between both states.

Recall that bounded model checking is usually applied to a well-defined unit of code, such as a function with specific conditions. The goal of bounded model checking is to qualitatively assert that no diversion from the specified postconditions is possible (i. e., any diversion implies a bug that must be fixed). In other words, the goal is to find a counterexample *violating* the postconditions. For the use case of synthesizing a gadget chain, the scenario is slightly different: There is no well-defined unit of code such as a function, but a large number of individual gadgets that can be executed in an arbitrary order. As a consequence, we are not interested in knowing whether specific

postconditions can be *violated* (as this most certainly is the case given the number and nature of the gadgets); instead, we are interested in whether there exists a chain of gadgets that *satisfies* the postconditions. In other words, we task the SMT solver with finding a satisfying assignment for $preconditions \land gadget_chain \land postconditions$. If the solver finds such an assignment, the produced model contains concrete values for all variables—including stack or other attacker-controlled buffers—which describe the chain of gadgets. Thus, once a model is found, converting the values into a chain becomes a trivial task. In the following, we present these steps in detail.

3.1 Gadgets

First, we must extract gadgets from the target program, which can then be further processed. This step is independent of the subsequent encoding and is covered in detail by previous works in this area [2, 18, 19, 20, 21]. As such, we omit it here for brevity. Note that we do not require the gadget extraction to be exhaustive or classify gadgets, as long as these sequences of instructions end with an indirect control-flow transfer. As assembly instructions commonly have side effects (e. g., `mul rbx` implicitly modifies the `rdx`, `rax`, and `rflags` register), we disassemble and lift the gadgets to an *intermediate representation (IR)* with explicit side effects. An example for two gadgets is visible in Figure 1a. Noteworthy, each IR instruction has no implicit side effects. We reiterate that—other than most state-of-the-art tools—our design imposes no restrictions, ranking, or classification on the gadgets.

3.2 Logical Encoding

Given a pool of gadgets, we want to query an SMT solver to find a chain of gadgets that transitions the initial program state (formulated as preconditions) into the desired program state (formulated as postconditions). For this, we need to logically encode the semantics of gadgets and chains. Especially, we must model the semantics of gadgets, the data flow between instructions, and the data flow between gadgets. Once we have encoded all components, we must combine them into a single formula, which we then pass to an SMT solver. To construct such a formula, we connect each statement through conjunctions. In the following, we first describe how individual gadgets are encoded and then explain how gadgets are interconnected to form a chain.

```
1  gadget_a:
2    mov rbx, [rsp+8] ; rbx := @64[rsp + 8]
3    mov [rsp], rdx   ; @64[rsp] := rdx
4    ret              ; rsp := rsp + 8
5                     ; rip := [rsp - 8]
```

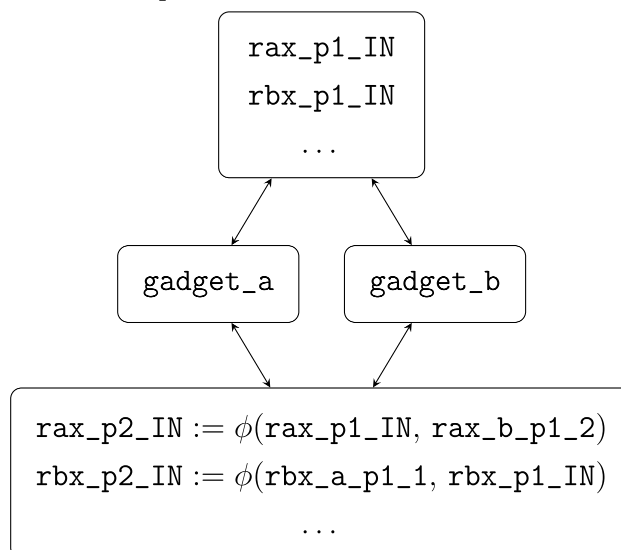
```
1  gadget_b:
2    pop rax ; rax := @64[rsp]
3            ; rsp := rsp + 8
4    inc rax ; rax := rax + 1
5    jmp rbx ; rip := rbx
```

(a) Assembly code and the corresponding intermediate representation (IR) of the instructions as comments. Note that side effects are explicitly modeled in the IR, thus a single assembly instruction may result in multiple IR instructions.

```
1  gadget_a:
2    rbx_a_1 := read(M_IN, rsp_IN + 8, 64)
3
4    M_a_1 := write(M_IN, rsp_IN, rdx_IN, 64)
5
6    rsp_a_1 := rsp_IN + 8
7    rip_a_2 := read(M_a_1, rsp_a_1 - 8)
```

```
1  gadget_b:
2    rax_b_1 := read(M_IN,
3                    rsp_IN, 64)
4    rsp_b_1 := rsp_IN + 8
5
6    rax_b_2 := rax_b_1 + 1
7    rip_b_2 := rbx_IN
```

(b) SSA form of the IR representation. The variable's locality is specified by an unique identifier, here `_a` or `_b`. Suffix `_IN` represents the initial definition.



(c) Structural overview of the final SMT formula, assuming a chain of two gadgets.

Figure 1: The high-level idea of our logical encoding: We lift assembly gadgets to an intermediate representation, make the variable and memory accesses stateful (via static single assignment form) and encode the data flow between gadgets using ϕ-functions.

Instructions and Gadgets. To use a gadget in the logical formula, we must first model all implicit state transitions on the instruction level: While a CPU executes a sequence of instructions in a row, it implicitly tracks state changes in registers and

memory. To represent this behavior in a logical formula, we must explicitly model it on the instruction and inter-instruction level. To address the instruction level, recall that we lift instruction into an IR form that explicitly handles side effects. For the latter, we have to model the data flow between instructions, e. g., when a register is assigned to another register or is defined more than once. To achieve this, we make variable assignments stateful by converting IR instructions into *static single assignment (SSA)* form [22]. This implies that each variable *definition* is assigned a new unique index, while *uses* always use the last defined index. To differentiate between gadgets, we prefix SSA variable names with an identifier that is unique to each gadget. If a gadget uses a variable that was not defined previously within this particular gadget, we postfix it by `_IN` to indicate that the value has been defined outside of the gadget's scope. In other words, it is an input to the gadget.

Example 1: *Figure 1b shows how* `rip_a_2` *depends on the memory at address* `rsp_a_1 - 8` *(line 7), which itself can be calculated as* `rsp_a_1 = rsp_IN + 8` *(line 6). Note the identifier* `_a`*, which distinctly marks this variable as belonging to gadget* `a`*, and the postfix* `_IN` *indicating that this instruction depends on* `rsp`*'s definition outside this gadget.*

Memory. Similar to registers, we apply SSA to memory to make it stateful, as otherwise, the SMT solver has no context information about memory addresses and values. To transform memory into SSA form, we define memory read and write accesses as explicit operations: `v_j := read(M_i, address, size)` and `M_i+1 := write(M_i, address, value, size)`. Given a stateful memory variable `M`, we read from and write to this variable at a given address with a given access size. Note that the write operation is stored in a new memory variable `M_i+1` that encodes the previous write. Internally, these operations are expressed within a byte-wise memory model similar to the work of Sinz et al. [17], in which memory accesses with larger sizes are translated to nested byte-wise memory reads or writes. For a formal definition, we refer the interested reader to Appendix A. We initialize all memory addresses to contain the value 0.

Interconnecting Gadgets. Up until now, we described how to encode data flow within a single gadget using SSA for registers and memory. However, our goal is to combine multiple gadgets in a chain of length n without making assumptions on neither the order of gadgets nor the particular gadgets used. Especially, we allow gadgets to occur more than once in the chain. Thus, in the next step, we have to logically

encode the data flow between gadgets. To achieve this, we first have to ensure that all variables are unique. So far, variables are only unique with respect to their gadget due to the SSA form's unique identifier. However, to encode the order of execution, each variable must also be unique with regard to the gadget's position within the chain. Therefore, we also include the position as index within the SSA variable names: `variable_gadgetId_position_definitionIdx`. This way, we can use any gadget at any position in the chain.

Example 2: *If we consider the gadget for the first position in the chain, the definition* `rbx_a_1` *(line 2 in Figure 1b) becomes* `rbx_a_p1_1` *(with* `p1` *representing the first position). This way, we can use the gadget in position 2 as well, as* `rbx_a_p2_1` *is a distinct variable.*

Naturally, our encoding must consider that a gadget can be used at any position in the chain, while, at the same time, we cannot know which gadget is at a specific position within the chain. In other words, `gadget_a` and `gadget_b` can both be at positions 1 and 2, but at the time of formula generation, we do not know which of these gadgets will be at which position in the chain synthesized by the SMT solver. Therefore, we must ensure that the gadget at position $i + 1$ uses the values derived by the gadget at position i; a scenario strikingly similar to the problem of merging control flow in SSA form (for which ϕ-functions are used). We must merge the state of all gadgets at chain position i such that it can be used as input for the gadgets in the subsequent position. To achieve this, we apply the following for each register and memory variable: We first determine the variable's last definition in each gadget for position i. Then, we merge the last definitions from all gadgets via a ϕ-function and define a new variable that is used as input for the next position.

Example 3: *Assume that we want to encode the gadgets for a chain of length 2 (cf. Figure 1c). For each register, we create a ϕ-function that merges the last definitions of these variables. In the following, we consider this process exemplary for* `rax` *at position 1. The initial value of* `rax` *is* `rax_p1_IN`*—the input of* `rax` *for the first gadget position. Since we do not know if* `gadget_a` *or* `gadget_b` *is the first gadget in the chain, we must account for both possibilities and merge their last definitions of* `rax` *in a ϕ-function.* `gadget_a` *does not modify* `rax`*, thus we use* `rax_p1_IN`*; for* `gadget_b`*, we use its latest definition,* `rax_b_p1_2`*. Finally, we define a new variable—*`rax_p2_IN`*—that encodes the merged variables and is used as input for the second position in the chain:* `rax_p2_IN` $:= \phi($`rax_p1_IN`, `rax_b_p1_2`$)$.

To model the data flow between gadgets, the logical formula has to connect each input variable of the ϕ-function with the gadget that defined the corresponding variable. On a technical level, we translate this abstract ϕ-function into nested `If-Then-Else` expressions that select the corresponding variable based on the program counter, which has to be equal to one of the gadget addresses. This way, we ensure that the conditions are mutually exclusive (as the program counter can only point to a single gadget) and, thus, each register's value can always be uniquely determined. This approach is based on work by Sinz et al. [17].

3.3 Preconditions and Postconditions

Following the logical encoding of the gadget chain, we now describe how to set the initial state (preconditions) and the targeted state (postconditions).

Preconditions. These conditions allow setting the initial state at the time when the attacker can divert execution flow to the gadget chain. They constraint the inputs of the first position in the gadget chain, e. g., we can encode relevant context from the target program, such as the value of specific registers or memory areas (e. g., by using a debugger). Additionally, we must specify the location where the SMT solver should place the synthesized gadget chain (and how many bytes are available), e. g., by choosing an attacker-controlled buffer on the stack. This area is then considered a free variable in the formula, such that the SMT solver can place gadget addresses and data there. We can also enforce specific characteristics for any attacker-controlled areas, such as constraining memory buffers to hold only values within a certain range.

Postconditions. While the preconditions outline the initial position, postconditions describe the desired state that the program should reach after executing the gadget chain. More specifically, we can set any register or memory address to a specific value (e. g., the system call we would like to execute and its arguments). We encode these postconditions by asserting that the outputs (i. e., register and memory variables) of the last position in the chain are equal to the given values.

Furthermore, we also support indirect constraints, so-called *pointer constraints*. These constraints support common constructs, where a reference to a specific value or string (e. g., "/bin/sh") in memory needs to reside in a specific register. To this end, we add an assertion that the memory address pointed to by this register must contain the desired value(s). This does not require us to specify the memory address itself, but we can leave

the task of choosing a suitable memory address to the SMT solver. On a technical level, the values are constrained as byte-wise memory read operations relative to the address chosen by the solver.

Notably, the flexibility of our approach allows us to enforce arbitrary constraints between registers and memory locations. For instance, we could enforce that (1) certain register values must be odd, (2) the sum of registers must be equal to a specific value, or (3) the sum of two specific registers must be prime. To put it differently, our design allows to constraint exotic, target-specific conditions that may be useful in some exploitation scenarios.

3.4 Formula Generation

Our final formula consists of three main components: preconditions, gadget chain, and postconditions. The preconditions describe the initial state, which is used as input for the chain's initial gadget. The chain contains the encoding of individual instructions, the data flow between instructions within a gadget, and the data flow between gadgets—in short, the complete semantics of the gadget chain. Finally, the postconditions define the state which should be reached after executing the gadget chain. Here, the attacker encodes the desired CPU state. We combine these three components with logical conjunctions to the formula:

$$formula := preconditions \land gadget_chain \land postconditions$$

We then pass this formula to an SMT solver that supports the combined quantifier-free theory of fixed-size bit vectors (registers) and arrays (memory), `QF_ABV` [23]. If the solver finds a satisfying assignment, it provides a model, i. e., concrete values for each relevant variable in the formula. For all variables of gadgets that are *not* relevant for the synthesized gadget chain, no values are assigned. As a consequence, the model describes not only the initial state (e. g., values on the stack) but register and memory values for each gadget in the chain; in other words, we receive sort of an instruction trace that includes the intermediate values for each variable in the chain. In a final step, we can extract the initial values for each controlled buffer and use them as exploitation payload. When the payload is inserted, the gadget chain is executed as described in the model. Because a satisfying assignment produced by an SMT solver is a proof of existence, the gadget chain is guaranteed to reach exactly the specified postconditions.

This is in strong contrast to state-of-the-art approaches, which often use heuristics rather than proofs to construct a gadget chain.

3.5 Algorithm Configuration

A few parameters define the performance of our approach, most of which affect the SMT solver: (1) For larger numbers of gadgets, the SMT solver needs more time in its decision process. To reduce its runtime, we can sample a small subset of gadgets (e. g., 300 gadgets as determined in empirical tests). (2) Due to our logical encoding, the chain length must be defined beforehand. While this may appear inflexible, our evaluation shows that testing different chain lengths is feasible in practice; if a shorter chain is possible, the SMT solver places semantic no-operations as padding gadgets in the chain. (3) To avoid excessive runtimes, we define upper time limits for the initial gadget extraction as well as for the SMT solver. While limiting the initial gadget extraction may reduce the number of available gadgets, this has no major impact if we only sample a subset.

4 Implementation

To demonstrate the practical feasibility of our proposed approach, we implemented a prototype of `SGC` in roughly 5,000 lines of Python code (see `https://github.com/RUB-SysSec/gadget_synthesis`). While `SGC`'s initial gadget extraction is based on `Binary Ninja` [24] (version 2.3.2660), all further steps are built on top of `Miasm` [25] (commit 218492cd). Especially the logical encoding of gadgets is facilitated in `Miasm`'s IR. We extended its internal memory model to be stateful. The logic formula generated in the encoding step is then passed to the SMT solver `Boolector` [26], which is particularly suited to solve problems within the domain theory of bit vectors and arrays [27]. As `Boolector` supports the `const-array` extension [28], we use it to model memory and initialize it with a default value of 0. As memory accesses should not happen in read/write-restricted regions, we allow the user to specify which memory addresses may be accessed. In general, the user can add any constraint they need, such as excluding specific bytes from the chain (so-called *bad bytes*).

5 Evaluation

Based on the prototype implementation of SGC, we answer the following questions:

1. Is SGC capable of automatically finding valid gadget chains in diverse exploitation scenarios? How does it compare to state-of-the-art tools?
2. How does SGC perform in real-world exploitation scenarios?
3. How flexible and target-specific are SGC's chains in comparison to other approaches?
4. In what regard do SGC's generated gadget chains differ from the ones found by state-of-the-art tools?

To answer these research questions, we conduct the following experiments.

5.1 Setup

All our experiments were performed using Intel Xeon Gold 6230R CPUs at 2.10 GHz with 52 cores and 188 GiB RAM, running Ubuntu 20.04 on `x86-64`. To facilitate a deterministic analysis, we disable ASLR. Even if present, we only require an attacker to leak the base address, e. g., via an information leak, which is a weaker requirement than other approaches make [5, 6].

We compare SGC against the state of the art discussed in Section 2. While these tools work deterministically and take all gadgets into account, SGC does not: To keep the runtime of the SMT solver manageable, a subset of gadgets is randomly sampled for a provided seed. As a consequence, the sampled gadgets may be insufficient to fulfill the attacker's goals. To mitigate this problem, SGC uses by default ten different seeds, running them in parallel and reporting the first chain found. To add further variety, SGC attempts to find a chain of length 3 and 5, both for 100 and 300 gadgets, while not using more than 128 bytes of the attacker-controlled buffer. These values have been empirically chosen (cf. Section 5.6) In summary, 40 configurations are executed in parallel. For our evaluation, we run all configurations until completion for later analysis instead of returning the first gadget chain found. As all other tools operate deterministically, we only run them once. We emphasize that all tools are provided equal resources, i. e., CPU cores and RAM. While we restrict SGC to one hour for

disassembly and the SMT solver, we define a timeout of 24 hours for all other tools. To verify whether a generated chain is *valid*, we use `GDB` to place it in the attacker-controlled buffer within the program and then execute the chain. This way, we ensure that the gadget chain works in practice.

As targets, we use a diverse set of programs. In a first step, we replicate the experiments of Follner et al. [14] on recent versions of `chromium` (version 88.0.4324.182), `apache2` (version 2.4.46), `nginx` (version 1.19.9), and `OpenSSL` (version 1.1.1f). All of these targets are dynamically linked and we configure `SGC` to ignore shared libraries, simulating a scenario where only the base address of the main executable is known but no locations of libraries. To cover scenarios where `libc` is present, we create an empty wrapper program that is statically linked against `glibc` version 2.31. To evaluate whether `SGC` can be used to exploit real-world vulnerabilities, we use `dnsmasq` (version 2.77).

5.2 Finding a Chain

Based on the experiments by Follner et al. [14], we evaluate whether `SGC` is capable of finding valid gadget chains. While a multitude of possible attacker goals exists, in reality, attackers mostly aim at either calling library functions such as `mprotect` (to change the protection flags of memory regions) and `mmap` (to map a RWX page in which their shellcode can be placed), or at executing system calls, such as `execve` with the parameter `/bin/sh` that spawns a shell. Therefore, we pick three exemplary attacker goals, namely (1) a library call to `mprotect(addr, len, prot)` with three parameters, (2) a library call to `mmap(addr, length, prot, flags, fd, offset)` with six parameters, and (3) a system call to `execve(path, argv, envp)` with four parameters (one being the system call number) and the requirement to place a string in memory. On the `x86-64` architecture, these arguments are passed via registers [29]. As parameters, we use fixed exemplary values that are common in real-world exploitation scenarios, such as `execve(&"/bin/sh", 0, 0)` to spawn a shell or setting `prot` in `mprotect` to `RWX`, such that an attacker could place and execute arbitrary shellcode. To compare the tools, we run each of them in the same configuration, analyze whether it finds a chain, and check—based on our verification tooling—if the chain is *valid* in practice. Table 2 depicts the results of this experiment. As `ROPgadget` only provides

Table 2: Capability of finding a valid gadget chain to call `mprotect`, `mmap`, or `execve`. Legend: ✓ = valid chain, (✓) = chain found but crashes program, ✗ = no chain found, [1]) = chain found when increasing timeout to 5h, [2]) = `SGC` *proves* that no chain exists.

		SGC	P-SHAPE	angrop	ROPium	ROPgadget	Ropper
mprotect	chromium	✓	✗	✗	✓	-	✗
	apache2	✓	(✓)	✓	✓	-	(✓)
	nginx	✓	(✓)	✓	✓	-	✗
	OpenSSL	✓	(✓)	✗	✗	-	✗
	libc	✓	(✓)	✓	✓	-	✓
mmap	chromium	✓[1]	✗	✗	✓	-	-
	apache2	✓	✗	✗	✓	-	-
	nginx	✓	(✓)	✗	✗	-	-
	OpenSSL	✗[2]	✗	✗	✗	-	-
	libc	✓	(✓)	✗	✓	-	-
execve	chromium	✓	-	✗	✓	✓	✗
	apache2	✓	-	(✓)	✓	✗	(✓)
	nginx	✓	-	(✓)	✓	✗	✗
	OpenSSL	✓	-	✗	✗	✗	✗
	libc	✓	-	✓	✓	✓	✓

fixed heuristics for `execve`, we exclude it from the other attacker goals. Similarly, `Ropper` is limited to `mprotect` and `execve`, and `P-SHAPE` focuses on library calls.

Most tools find a chain for `mprotect`, which is the easiest goal since only three registers have to be set. `angrop` struggled both with `chromium` and `OpenSSL` and crashed during the attempt to locate gadget chains. Likewise, `P-SHAPE` crashed for `chromium`. Although `P-SHAPE` found a chain for four targets, none of them were valid in real-world scenarios: Manual verification revealed that they cause segmentation faults (e. g., due to write attempts to inaccessible memory regions). For `mprotect`, only `SGC` identifies a valid gadget chain for all targets.

In comparison to `mprotect`, finding a chain for `mmap` is significantly more challenging since six register arguments have to be set, and thus more suitable gadgets are required. While all chains found by `P-SHAPE` crashed again, `ROPium` produced valid chains for

three targets. However, this was only possible after we fixed a bug in its source code. SGC found four out of five valid chains. For chromium, we had to increase the timeout for disassembly and solving to 5h, since we initially did not find suitable gadgets to set r8 and r9, the fifth and sixth argument to mmap. We discuss the shortcomings of our disassembly and random sampling in more detail in Section 6. For OpenSSL, no tool was able to produce a chain. To get more insights, we performed another experiment in which SGC was given access to all 3045 available OpenSSL gadgets (instead of choosing a random subset). After 226*s*, the SMT solver returned UNSAT, which can be understood as proof of non-existence. In other words, SGC was able to assert that no chain for the provided gadgets exists that fulfills the postconditions. This saves the user valuable time as they are guaranteed that even manual analysis will be fruitless.

The last attacker goal, execve, models the common scenario where a shell is spawned via a system call. It differs from the previous goals in the fact that not only four register values must be prepared, but the string /bin/sh\x00 must be placed in memory. To express this behavior in ROPium, the user has to manually set a suitable memory address at which the string should be placed in memory. As such, the gadget chain construction is not completely automated. However, we include it since it is the only tool besides SGC that succeeds in finding valid chains for almost all targets.

In summary, these experiments answer research question 1: SGC outperforms all state-of-the-art approaches and manages to find valid gadget chains for all targets, even when other tools fail. For the only case where it did not find a chain, it even provided formal proof that no chain for the available gadgets can exist.

5.3 Real-World Applicability

To answer research question 2, we are interested in whether SGC proves helpful towards finding gadget chains in real-world exploitation contexts. To this end, we conduct a case study for CVE-2017-14493 [30], which describes a stack-based buffer overflow in dnsmasq (up to version 2.77) [31]. In essence, an attacker can craft a malicious DHCPv6 packet that, when received by dnsmasq's DHCP server, triggers an overflow in the dhcp6_maybe_relay function, where the length and data of a memcpy can be controlled by the attacker. This bug allows for the injection of gadget chains of arbitrary length; if ASLR is present, an attacker can exploit an information leak in the same version,

assigned CVE-2017-14494, to leak the base address [30]. For simplicity, we assume ASLR is already bypassed.

Our goal is to craft a gadget chain that calls `execve(&"/bin/sh", 0, 0)` to spawn a shell. Following the System V AMD64 ABI calling convention [29], register `rax` needs to hold the `execve` system call number (`0x3b`), while the registers `rdi`, `rsi`, and `rdx` pass the arguments to `execve`. Therefore, we set the postconditions accordingly. To define the preconditions, we have to inspect the program state at the time when the attacker can divert execution flow to the gadget chain. In detail, we dump the CPU state with `GDB` and constraint register values accordingly. After defining preconditions and postconditions, we logically encode the gadget chain and query the SMT solver with the formula. `SGC` finds a gadget chain after approximately 8m. A shell is spawned after embedding the gadget chain in a DHCPv6 packet and sending it to `dnsmasq` For a detailed explanation of the bug and chain found by `SGC`, we refer to Appendix B. To conclude research question 2, `SGC` assists in real-world exploitation scenarios. It only requires the initial CPU state as preconditions and the desired target state.

5.4 Target-Specific Constraints

To answer research question 3 that addresses the flexibility of our approach, we conduct two experiments that model different exploitation scenarios. In the first experiment, we aim at crafting chains that do not include so-called *bad bytes*. Such bytes cannot be used in an exploit payload since they act as terminators in the underlying program (e. g., `\x00` in C strings). We can avoid using such bytes in our payload by adding the constraint that each byte in the attacker-controlled buffer must be different from specific byte values. In this experiment, we try to craft valid gadget chains that call `mprotect`, `mmap`, and `execve` in the statically-linked `libc` wrapper, where `\x0a` and `\x0b` are considered as bad bytes. `SGC` produced a valid gadget chain within, on average, 512s; similarly, all other tools (excluding `P-SHAPE`, which does not support bad bytes) were able to produce gadget chains. This is not surprising, as avoiding bad bytes is a common requirement for many exploits and most tools consider this in their heuristics. Then, we slightly modify this experiment: We include one of the parameter values passed to the functions as a bad byte (essentially prohibiting the tools from using this specific value directly), such that the tools must construct the value indirectly via the

Table 3: Statistics over all valid chains generated during experiments in Section 5.2.

		SGC	P-SHAPE	angrop	ROPium	ROPgadget	Ropper
avg. instructions		5.9	-	2.9	2.4	2.0	2.6
gadgets w/ mem. write		9%	-	7%	6%	3%	14%
└ excluding execve		9%	-	0%	0%	-	0%
gadgets w/ mem. reads		30%	-	7%	0%	0%	0%
└ excluding execve		32%	-	0%	0%	-	0%
CF types	ret	68%	-	100%	97%	100%	100%
	call MEM	10%	-	0%	0%	0%	0%
	call REG	20%	-	0%	3%	0%	0%
	jmp REG	2%	-	0%	0%	0%	0%

gadget chain. In this scenario, only ROPium and SGC manage to find valid gadget chains. This shows that even a standard feature can be problematic for heuristics-based tools.

In the second experiment, we add a more complex constraint: We require that the sum of all values (quadwords) in the attacker-controlled buffer (where the addresses and data for the gadget chain are placed) must be equal to the value 0xdeadbeef. While this constraint seems artificial, similar constraints can be found in commercial DRM systems that perform integrity checks over specific memory regions. While no other tool provides the flexibility to model this behavior, we can enforce this within a few lines of code in SGC and produce valid gadget chains for the same setup as before (within, on average, 527s).

Overall, we conclude that SGC provides great flexibility and allows to model complex constraints. Thus, it covers even unusual exploitation scenarios.

5.5 Chain Statistics

To answer research question 4, in what regard differ our gadget chains from the ones found by state-of-the-art approaches, we inspect which types of gadgets and instructions are used in the generated chains. To this end, we analyze each valid chain found during our experiment in Section 5.2. Since P-SHAPE found only invalid chains that crashed the program, we exclude it from this experiment.

Table 4: SGC's timings for initial disassembly and chaining.

	Disassembly	Chaining	Total
mprotect	1845s	363s	2207s
mmap	1617s	2667s	4284s
execve	1845s	494s	2338s

As visible in Table 3, SGC's gadgets contain on average almost six instructions, whereas the other tools use two to three instructions per gadget. Further, SGC is the only approach that makes use of explicit memory reads and writes (excluding instructions such as push and pop); all other tools only use it in the case of execve to place the string /bin/sh into the memory. Similarly, most of the tools rely exclusively on return-oriented gadgets; only ROPium uses call-oriented programming for 3% of its gadgets. Contrary, SGC only uses return-oriented programming in 68% of the cases, while it deploys call and jump-oriented gadgets in 32%. In summary, SGC has on average longer gadgets, uses more memory reads/writes, and has a significantly higher amount of non-return-oriented gadgets; in short, it includes gadgets specific to the target with side effects that are disregarded by other approaches due to their generic heuristics.

Another relevant aspect is SGC's runtime (cf. Table 4). The disassembly step is comparably slow; the time required for instruction lifting, encoding, and SMT solving is significantly lower. Our disassembly relies on a combination of Binary Ninja and Miasm: we first analyze the whole binary and disassemble then individual functions in Miasm. As it is not a focus of this work, we consider improving our disassembly component as future work. Only for mmap, finding the chain takes significantly more time since the SMT solver has to find a valid chain that prepares six function arguments. For reference, the other tools find a chain on average within 319s. However, this ignores the runtime when they found no chain (e. g., Ropper hit the timeout of 24h twice), which was often the case, especially for mmap. In summary, SGC manages to find a valid chain within minutes.

5.6 SGC's Configuration

After successfully answering all research questions, we would like to give a better intuition of the configuration parameters relevant for SGC. As described before, our approach

Table 5: Number of gadget chains the solver decided (i. e., considered `SAT` or `UNSAT`) vs. timeouts when building a chain to `mprotect` for the targets in Section 5.2 with ten different seeds each. Format is `#Decided by SMT solver`/`#Timeout`. We color the prevalent outcome.

#Gadgets	Chain Length 1	2	3	4	5	6	7	8
100	50/ 0	50/ 0	49/ 1	31/ 19	24/ 26	16/ 34	15/ 35	12/ 38
300	50/ 0	50/ 0	37/ 13	20/ 30	13/ 37	10/ 40	7/ 43	6/ 44
500	50/ 0	44/ 6	31/ 19	16/ 34	10/ 40	8/ 42	5/ 45	4/ 46
1000	50/ 0	31/ 19	25/ 25	11/ 39	9/ 41	2/ 48	0/ 50	0/ 50

is probabilistic: it randomly samples only a small subset of gadgets. As a result, the chosen subset may not be sufficient to generate a chain that fulfills the postconditions. We can select another subset of the same size or a larger number of gadgets to overcome this. The latter, however, increases the time required by the SMT solver to decide the chain synthesis problem. To get a better feeling for this trade-off, we vary the chain length and number of sampled gadgets and analyze how often the solver succeeds in deciding the synthesis problem, i. e., it finds a chain or returns `UNSAT` within one hour. For each configuration, we run the solver ten times with different seeds such that diverse gadgets are sampled. We do this for all target programs from Section 5.2 and count how often the solver finds an answer or timeouts in the process of finding chains for `mprotect`. In total, we perform 50 independent runs (ten different seeds for five different targets) for each configuration.

As Table 5 shows, the chain length and the number of gadgets determine the SMT solver's performance: For a small number of gadgets and chain length of 1, the solver always finds an answer. However, for longer chains or more sampled gadgets, the number of timeouts increases. While the solver can decide some chains of length six or higher, it increasingly triggers the timeout of one hour. Similarly, for a larger gadget pool (e. g., 1000 gadgets), the solver already struggles for chains of length three. While the strategy of randomly sampling a small number of gadgets proved effective, an attacker can always increase the number of gadgets and set higher timeouts for the SMT solver.

6 Discussion

Limitations of `SGC`. While `SGC` has proven overall effective, various aspects can be improved: (1) Our currently used disassembly is naive since we only consider regular

instruction offsets. As an improvement, we can search unaligned gadgets since any sequence of bytes can be interpreted as instructions on `x86-64`. (2) The SMT solver is the most significant performance bottleneck of our design as it may require a large amount of time to identify valid gadget chains. However, as our evaluation shows, randomly selecting a subset of gadgets provides an effective strategy to reduce `SGC`'s runtime. In this scenario, an `UNSAT` provided by the SMT solver is *not* a formal proof that no gadget chain exists, as it only proves that no chain for the selected subset of gadgets exists.

Mitigations. To prevent exploitation, various mitigations have been proposed.

(1) `W^X` prevents execution of injected code, however, it is ineffective against *code reuse* attacks and thus `SGC`. (2) Address space layout randomization (ASLR) shuffles the program's memory layout such that an attacker cannot rely on addresses. `SGC` requires only the base address of the code section and does not require shared libraries to find valid gadget chains, thus a single information leak suffices. (3) Lastly, control-flow integrity (CFI) prevents the redirection of control flow to arbitrary code locations. This severely hampers code-reuse attacks such as `SGC` because only specific gadgets can be chained together. However, related work has shown that even fine-grained CFI is insufficient to prevent code-reuse attacks in general [6, 7]. We believe that an attacker could add constraints modeling the enforcement policies such that the SMT solver will only select gadget chains that pass the CFI enforcement policy. We leave this as interesting future work.

7 Related Work

After initial techniques in the domain of code-reuse focused on functions from `libc` [1], the concept was generalized to re-use small snippets of existing code [2, 32]. These small snippets are often chained via `ret` instructions (ROP) [2], but other control-flow transfers work as well (JOP [18, 19] and COP [20, 21]). Mitigations such as ASLR have been shown to be insufficient [33]. Moving forward with new mitigations such as control-flow integrity (CFI) [4], even more advanced approaches have been proposed, e. g., counterfeit object-oriented programming (COOP) [9] or data-oriented programming (DOP) [34]. Even fine-grained CFI solutions fail to stop attackers from finding gadget chains [5].

In parallel, various techniques to automate the cumbersome task of identifying suitable gadgets have been proposed. Early approaches use pattern matching to search for desired gadgets [35, 36]. Other approaches tackle the task of automating the attack itself: One of the earliest approaches, `Q` [8], uses software verification methods instead of pattern matching to achieve this goal. Using identification and chaining of gadgets similar to `Q`, Wollgast et al. [37] automate COP, which allows them to bypass coarse-grained CFI implementations. Tackling the problem imposed by fine-grained CFI solutions, Ispoglou et al. [6] propose an approach, `BOPC`, which automates data-only attacks. Further improving this avenue, Schwartz et al. [7] propose a generic approach, `Limbo`, capable of constructing chains using ROP, JOP, COP, or DOP. Their approach is similar to ours in the spirit of maintaining a generic approach to code-reuse attacks. However, their focus is on the construction of CFI-compatible gadget chains. Internally, their search relies on concolic execution and hard-coded heuristics. In contrast, our approach does not tackle the problem of identifying CFI-aware gadgets but maintains generality without relying on hard-coded heuristics. Further, `Limbo` only works for 32-bit Linux executables, which limits their real-world applicability. As no code is published, we cannot evaluate against `Limbo`.

8 Conclusion

In this paper, we presented a generic and flexible approach to automate the task of finding gadget chains. With our prototype implementation, we have shown that `SGC` outperforms state-of-the-art tools. It not only finds gadget chains where all other approaches fail but also allows to model complex constraints.

Acknowledgments

This work was supported by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States—EXC 2092 CaSa—39078197.

References

[1] Solar Designer. Return-to-Libc, 1997.

[2] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

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

[4] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1), 2009.

[5] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[6] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block-Oriented Programming: Automating Data-only Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[7] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari, and Stephanie M Schwartz. A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[8] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*, 2011.

[9] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[10] Jonathan Salwan. ROPgadget. `https://github.com/JonathanSalwan/ROPgadget`.

[11] Sascha Schirra. Ropper. `https://github.com/sashs/Ropper`.

[12] angr team. angrop. `https://github.com/angr/angrop`.

[13] Boyan Milanov. ROPium. `https://github.com/Boyan-MILANOV/ropium`.

[14] Andreas Follner, Alexandre Bartel, Hui Peng, Yu-Chen Chang, Kyriakos Ispoglou, Mathias Payer, and Eric Bodden. PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution. In *Security and Trust Management Workshop*, 2016.

[15] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT Solvers in Software Security. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[16] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2016.

[17] Carsten Sinz, Stephan Falke, and Florian Merz. A Precise Memory Model for Low-level Bounded Model Checking. In *International Conference on Systems Software Verification*, 2010.

[18] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[19] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming Without Returns. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[20] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security Symposium*, 2014.

[21] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[22] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1989.

[23] SMT-LIB. Logics. `https://smtlib.cs.uiowa.edu/logics-all.shtml#QF_ABV`.

[24] Vector 35 Inc. Binary Ninja. `https://binary.ninja/`.

[25] CEA IT Security. Miasm – Reverse Engineering Framework. `https://github.com/cea-sec/miasm`.

[26] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):53–58, 2014.

[27] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1), 2019.

[28] Aaron Stump, Clark W Barrett, David L Dill, and Jeremy Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *IEEE Symposium on Logic in Computer Science*, 2001.

[29] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[30] Fermin J. Serna, Matt Linton, and Kevin Stadmeyer. dnsmasq stack-based buffer overflow (CVE-2017-14493). `https://security.googleblog.com/2017/10/behind-masq-yet-more-dns-and-dhcp.html`.

[31] Simon Kelley. dnsmasq. `https://thekelleys.org.uk/dnsmasq/doc.html`.

[32] Sebastian Krahmer. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique, 2005.

[33] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[34] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy*, 2016.

[35] Ryan Glenn Roemer. Finding the Bad in Good Code: Automated Return-Oriented Programming Exploit Discovery. Master's thesis, UC San Diego, 2009.

[36] Tim Kornau. Return-Oriented Programming for the ARM Architecture. Master's thesis, Ruhr-Universität Bochum, 2010.

Table 6: Encoding of memory reads of various sizes, returning a value from memory m at address k.

Name	SMT encoding
$mem_read_8(m, k)$	$select(m, k)$
$mem_read_{16}(m, k)$	$concat(mem_read_8(m, k), mem_read_8(m, k + 1))$
$mem_read_{32}(m, k)$	$concat(mem_read_{16}(m, k), mem_read_{16}(m, k + 2))$
$mem_read_{64}(m, k)$	$concat(mem_read_{32}(m, k), mem_read_{32}(m, k + 4))$

Table 7: Encoding of memory writes of various sizes, returning a memory with value v at address k.

Name	SMT encoding
$mem_write_8(m, k, v)$	$store(m, k, v_{0:7})$
$mem_write_{16}(m, k, v)$	$mem_write_8(mem_write_8(m, k, v_{0:7}), k + 1, v_{8:15})$
$mem_write_{32}(m, k, v)$	$mem_write_{16}(mem_write_{16}(m, k, v_{0:15}), k + 2, v_{16:31})$
$mem_write_{64}(m, k, v)$	$mem_write_{32}(mem_write_{32}(m, k, v_{0:31}), k + 4, v_{32:63})$

[37] Patrick Wollgast, Robert Gawlik, Behrad Garmany, Benjamin Kollenda, and Thorsten Holz. Automated Multi-architectural Discovery of CFI-resistant Code Gadgets. In *European Symposium on Research in Computer Security (ESORICS)*, 2016.

A Memory Modeling

Byte-wise memory reads and writes are modeled using single *select* and *store* operators, respectively. Larger reads are modeled by concatenating multiple *select* expressions, which we define recursively in terms of smaller read operations. Reads smaller than 64-bit into a 64-bit register are zero-extended by using *concat* with the zero bit vector bv_0. Larger writes are similarly modeled using the composition of multiple *store* expressions. Table 6 and 7 show memory accesses of various sizes. Given an array m, address k and value v and bit size $n \in (8, 16, 32, 64)$, we use the names $mem_read_n(m, k)$ and $mem_write_n(m, k, v)$ to substitute the longer SMT expressions from these tables.

```
206     /* RFC-6939 */
207     if ((opt = opt6_find(opts, end, OPTION6_CLIENT_MAC, 3)))
208       {
209         state->mac_type = opt6_uint(opt, 0, 2);
210         state->mac_len = opt6_len(opt) - 2;
211         memcpy(&state->mac[0], opt6_ptr(opt, 2), state->mac_len);
212       }
```

Figure 2: Vulnerable `memcpy` in file `rfc3315.c` triggering the overflow of the `mac` buffer in struct `state`.

B dnsmasq CVE-2017-14493

In the following, we analyze the `dnsmasq` bug in more detail. The stack-based buffer overflow in `dnsmasq` is caused by the absence of a length check of the data copied to a static buffer on the stack. Figure 2 shows the vulnerable call to `memcpy` in function `dhcp6_maybe_relay`. Sending a malicious DHCPv6 packet allows an attacker to gain control over the instruction pointer by overflowing the `mac` buffer of static size `DHCP_CHADDR_MAX` (16) in the `state` structure present on the stack.

The proof-of-concept (PoC) provided alongside the bug report [30] builds up such a DHCPv6 packet containing an `OPTION6_CLIENT_MAC` option holding data of excessive length. While the PoC overwrites the instruction pointer with a dummy value, injecting an arbitrary amount of bytes is possible. As long as the stack is not exhausted, the packet's content is copied and remains untouched until the instruction pointer is overwritten.

In order to synthesize a gadget chain, the information needed to specify preconditions and postconditions is gathered by extracting the program state before hijacking the control flow through `GDB`. Table 8a shows the preconditions set for `dnsmasq`. The initial `ret` instruction, which redirects the control flow to the chain's first gadget (`gadget_0`), is specified by preconditioning `rip`. The stack pointer `rsp` points to the part of the controlled buffer, where the gadget chain will be copied. In the logical formula, this stack area is a free variable.

Since we want to execute a system call to `execve` to spawn a shell, the final register values which the gadget chain needs to reach are specified accordingly. Table 8b shows the postconditions in preparation for calling `execve(&"/bin/sh", 0, 0)`. Here, `rip` holds the address of a `syscall` instruction available in the program. Using the default

Table 8: Preconditions and postconditions used for `dnsmasq`. Registers not mentioned in the preconditions are free variables, i. e., registers an attacker controls and can set to an arbitrary value.

(a) Preconditions

Register	Value
rip	`0x33dfb`
rax	`0x223`
rcx	`0x0`
rdx	`0x5a`
rdi	`0x22`
r8	`0x7fffffffe0e0`
r9	`0x0`
r10	`0x7fffffffbc50`

(b) Postconditions

Register	Value
rip	`0x461d0`
rax	`0x3b`
rsi	`0x0`
rdx	`0x0`
rdi	`&"/bin/sh"`

configuration described in Section 5.1, `SGC` finds a gadget chain consisting of four gadgets within approximately $8m$. While most gadgets are straightforward, `gadget_3` (shown in Figure 3) writes a value to the stack outside the attacker-controlled buffer, a side effect that does not harm the chain. The arithmetic operations of the first four instructions do not change register `rax`' value of 0. In line 6, the `lea` instruction is used to add `0x5` to the value present in `rbp` = `0x55555559a1cb`. The resulting address, `0x55555559a1d0`, is a `syscall` instruction; the address is placed on the stack at address `0x7fffffffe240` present in register `rbx`. As this address is writable memory, no harm results from this side effect.

As mentioned earlier, the PoC crafts a rogue DHCPv6 packet. In order to construct the payload with our synthesized gadget chain, the length parameter is adjusted and the dummy value is replaced with the data of the gadget chain. Sending this packet to the `dnsmasq` DHCP server successfully spawns the shell.

```
1  0x55555558a009:
2      movzx     rax, ax
3      imul      rax, ax, 0x1DCB
4      shr       eax, 0x15
5      movzx     eax, ax
6      lea       rax, qword ptr [rax + rbp + 0x5]
7      mov       qword ptr [rbx], rax
8      pop       rbx
9      pop       rbp
10     pop       r12
11     ret
```

Figure 3: `gadget_3` of the gadget chain used to spawn a shell in `dnsmasq`.

Loki: Hardening Code Obfuscation Against Automated Attacks

Publication Data

Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening Code Obfuscation Against Automated Attacks. In *USENIX Security Symposium*, 2022.

Permission. *Reprinted with permission.*

Loki: Hardening Code Obfuscation Against Automated Attacks

Moritz Schloegel[1], Tim Blazytko[1], Moritz Contag[1], Cornelius Aschermann[1], Julius Basler[1], Thorsten Holz[2], Ali Abbasi[1],

[1] Ruhr University Bochum
[2] CISPA Helmholtz Center for Information Security

Abstract

Software obfuscation is a crucial technology to protect intellectual property and manage digital rights within our society. Despite its huge practical importance, both commercial and academic state-of-the-art obfuscation methods are vulnerable to a plethora of automated deobfuscation attacks, such as symbolic execution, taint analysis, or program synthesis. While several enhanced obfuscation techniques were recently proposed to thwart taint analysis or symbolic execution, they either impose a prohibitive runtime overhead or can be removed in an automated way (e. g., via compiler optimizations). In general, these techniques suffer from focusing on a single attack vector, allowing an attacker to switch to other, more effective techniques, such as program synthesis.

In this work, we present Loki, an approach for software obfuscation that is resilient against all known automated deobfuscation attacks. To this end, we use and efficiently combine multiple techniques, including a generic approach to synthesize formally verified expressions of arbitrary complexity. Contrary to state-of-the-art approaches that rely on a few hardcoded generation rules, our expressions are more diverse and harder to pattern match against. We show that even the state-of-the-art approach on Mixed-Boolean Arithmetic (MBA) deobfuscation fails to simplify them. Moreover, Loki protects against previously unaccounted attack vectors such as program synthesis, for which it reduces the success rate to merely 19%. In a comprehensive evaluation, we show that our design incurs significantly less overhead while providing a much stronger protection level compared to existing works.

1 Introduction

Obfuscation describes the process of applying transformations to a given program with the goal of protecting the code from prying eyes. Generally speaking, obfuscation works by taking (parts of) a program and transforming it into a more complex, less intelligible representation, while at the same time preserving its observable input-output behavior [1]. Usually, such transformations come at the cost of increased program runtime and size, thus trading intelligibility for overhead. Although formal verification of code transformations is hard to achieve in practice [2, 3], obfuscation is used in a wide range of real-world scenarios. Examples include protection of intellectual property (IP), digital rights management (DRM), and concealment of malicious behavior in software. Generally speaking, obfuscation protects critical (often small) code parts against reverse engineering and, thus, misuse by competitors or other parties. For example, most contemporary DRM systems rely on some kind of obfuscation to prevent attackers from distributing unauthorized copies of their product [4]. License checks and cryptographic authentication schemes are examples for code that is commonly obfuscated in practice to prevent analysis. Most copy-protection schemes used by games use some kind of obfuscation to prevent unauthorized copies. As another example, market-leading companies, such as *Snapchat*, obfuscate how API calls to their backend are constructed, preventing abuse and access by competitors [5].

Among the countless obfuscation methods proposed in the literature [1, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20], one of the most promising techniques is Virtual Machine (VM)-based obfuscation [9, 21]. State-of-the-art, commercial obfuscators such as Themida [22] and VMProtect [23], as well as most game copy-protection schemes used in practice [24, 25], make extensive use of VM-based obfuscation. They transform the to-be-protected code from its original Instruction Set Architecture (ISA) into a *custom* one, and bundle an interpreter with the program that emulates the new ISA. This effectively breaks any analysis tool unfamiliar with the new architecture. Attackers aiming to deobfuscate code affected by this scheme must first uncover the custom ISA before they can reconstruct the original code [21, 26]. Since the custom instruction sets are conceptually simple, VM-based obfuscation software usually applies additional obfuscating transformations to the interpreter such that it is harder to analyze. For example, dead code insertion or constant unfolding are often used. At their core, these

transformations inflate the number of executed instructions and mainly add to the code's *syntactic* complexity, but can be successful in thwarting manual attacks.

However, it is often sufficient to apply well-known compiler optimizations, such as *dead code elimination*, *constant folding*, or *constant propagation*, to reduce the code's *syntactic* complexity and enable subsequent analyses [27, 28]. We tested this hypothesis and observe that this applies to the state-of-the-art tools Themida and VMProtect, for both their fastest and strongest protection configurations: We found that a simple dead code elimination manages to reduce the number of assembly instructions per handler by at least 50% for five different targets, tremendously simplifying both manual and automated analyses (cf. Table 1). Subsequently, the resulting code can be further simplified using a wide range of automated techniques, including taint analysis [27, 29], symbolic execution [27, 30], program synthesis [31, 32], and various others [21, 26, 28, 33, 34, 35, 36, 37, 38, 39, 40, 41].

The reliance on *syntactic* complexity in state-of-the-art obfuscation schemes and the broad arsenal of advanced deobfuscation techniques sparked further research in the construction of more resilient schemes that aim to impede these automated analyses. Proposals were made to hinder taint analysis [42, 43] and render symbolic execution ineffective [6, 12, 15, 16]. For example, the latter can be achieved by triggering a path explosion for the symbolic execution engine by artificially increasing the number of paths to analyze. Other promising obfuscation schemes emerged, including Mixed Boolean-Arithmetic (MBA) expressions [6, 39, 44] that offer a model to encode arbitrary arithmetic formulas in a complex manner. The expressions are represented in a domain that does not easily lend itself to simplification, effectively hiding the actual semantic operations. Usually, automated approaches to deobfuscate MBAs are based on symbolic simplification [35, 36, 37, 39]; they rely on certain assumptions about the expression's structure, making them unfit to simplify such expressions in the general case. Other approaches are based on program synthesis [31, 32, 45], which have been proven highly effective for most tasks. In general, synthesis-based deobfuscation techniques remain unchallenged to date and are valuable methods for automated analysis of obfuscated code. Recent works aiming at simplifying MBA turned towards machine learning [46] and algebraic simplification [47]. Especially the latter approach, relying on a hidden two-way feature between 1-bit and n-bit variables used within MBAs, provides an automated attacker with unprecedented MBA deobfuscation capabilities.

Table 1: VM handler statistics for Themida, VMProtect, and our approach called Loki. The two commercial obfuscators are configured in their fastest (*Virtualization* and *Tiger White*) and strongest configuration (*Ultra* and *Dolphin Black*), but without additional security features (e. g., anti-debug). We track their handlers' *average number* of assembly and intermediate language (IL) instructions before and after dead code elimination (denoted as the percentage-wise reduction in parentheses). All values are averaged over five cryptographic algorithms (AES, DES, MD5, RC4, and SHA-1).

	VMProtect		Themida		Loki
Statistics	Virtualization	Ultra	Tiger White	Dolphin Black	
Assembly instructions	69 (−50.79%)	73 (−51.58%)	219 (−53.68%)	243 (−56.01%)	222 (−1.14%)
IL instructions	75 (−50.88%)	80 (−51.89%)	221 (−53.76%)	247 (−55.94%)	234 (−1.44%)
Handlers executed	46,591	151,303	83,191	290,815	4,123
. . . of them unique	274	4,578	204	337	55

In this paper, we introduce a novel and comprehensive set of obfuscation techniques that can be combined to protect code against all known automated deobfuscation attacks, while imposing only reasonable overhead in terms of space and runtime. Our techniques are specifically designed such that a human analyst gains no significant advantage from employing automated deobfuscation techniques, including compiler optimizations (cf. Table 1), forward taint analysis, symbolic execution, and even program synthesis (cf. Section 6). We explicitly assume scenarios where these techniques are specifically tailored to our design *(white-box scenario)*.

To achieve such protection, we propose a generic algorithm to synthesize formally verified, arbitrarily complex MBA expressions. This is in strong contrast to state-of-the-art approaches that rely on a few handwritten rules, greatly limiting their effectiveness. For example, given 7,000 VM handlers, Tigress—the state-of-the-art academic obfuscator—uses only 16 unique MBAs, while our design features ~5,500 unique MBAs. As a result, our MBAs are highly unlikely to be simplified: In fact, current state-of-the-art MBA deobfuscation tools such as MBA-Blast [47] can only simplify 0.5% of Loki's MBAs. Furthermore, we conduct the first conclusive analysis of the limits of program synthesis with regard to deobfuscation. Based on the resulting insights, we present a hardening technique capable of impeding program synthesis, reducing its success rate to 19%—for Tigress, it is 67%. In summary, we present a new design featuring both high diversity and resilience against static and dynamic, automated deobfuscation attacks. While providing more value, our design incurs significantly less overhead com-

pared to commercial, state-of-the-art obfuscation schemes (up to 40 times). Moreover, we port modern testing techniques, i. e., formal verification and fuzzing, to our design and show that complex combinations of obfuscation transformations benefit from such methods to assert the correctness of complex and non-deterministic obfuscation transformations.

Contributions. We make the following contributions:

- We present the design, implementation, and evaluation of Loki, a software obfuscation approach resilient against all known automated deobfuscation attacks, even in white-box scenarios.
- We introduce a generic approach to synthesize diverse and formally verified Mixed Boolean-Arithmetic (MBA) expressions of arbitrary complexity that withstand even current state-of-the-art deobfuscation attacks.
- We are the first to propose an approach resilient against program synthesis-based attacks and map out limits of program synthesis in an empirical evaluation.

We publish the source code of Loki as well as all evaluation artifacts (including test cases, binaries, and evaluation tooling) at `https://github.com/RUB-Syssec/loki`. An extended version of this paper with more technical details is available as a technical report [48].

2 Technical Background

We start by providing an overview of the required technical information on obfuscation and deobfuscation techniques.

2.1 VM-based Obfuscation

Virtual machine-based obfuscation, also known as *virtualization*, protects code by translating it into an intermediate representation called *bytecode*. This bytecode is interpreted by a CPU implemented in software, adhering to a custom instruction set architecture (ISA). An attacker must first reverse engineer this software CPU, a tedious and time-consuming task [21, 26]. Only after understanding the VM, they can reconstruct the original high-level code.

VM Interpreter. The original, unprotected code is replaced with a call to the *VM entry* that invokes the interpreter. It sets up the initial context of the VM and points it to the bytecode that is to be interpreted. This is implemented by the *VM dispatcher* using a fetch-decode-execute loop: first, it fetches the next instruction, decodes its opcode, and then transfers execution to the respective *VM handler*. Often, the handler is determined via a *global handler table* that is indexed by the opcode. After handler execution, the control flow returns to the VM dispatcher. Eventually, execution finishes by invoking a special *VM exit* handler aborting the loop.

Abstraction of Handler Semantics. Handlers are often semantically simple [21, 31]; they perform a single arithmetic or logical operation on a number of operands, e. g., $x \odot y$. We call the semantic function of a handler, i. e., the underlying instruction it implements, its *core semantics*. We can represent core semantics as a function $f(x, y)$, or more general as $f(x, y, c)$ where c is a constant. To measure the *syntactic complexity* of the core semantics, we compute the (syntactic) expression depth of f as the sum of all variable occurrences and operators. In contrast, the *semantic depth* refers to the syntactic depth of the syntactically shortest equivalent expression. Intuitively, it can be understood as the number of nodes in an Abstract Syntax Tree (AST).

Example 4: *We can represent a VM handler's core semantics* $x+y$ *as* $f(x, y, c) := x+y$ *with a syntactic depth of* 3*. A syntactically more complex function* $g(x, y, c) := x + y - x + c - c$ *has a syntactic depth of* 9 *but a semantic depth of* 1*, since* g *can be simplified to* $g(x, y, c) := y$*.*

Superoperators. Superoperators [49] are an approach to make handlers semantically more complex. Intuitively, this is achieved by combining different instruction sequences from the unprotected code into a single VM handler. Usually, these sequences compute independent results such that this VM "superhandler" computes multiple, independent VM handlers in a single step. As a consequence, superoperators often have multiple input and output tuples. Related to our function abstraction, we can say the function $f_s((x_0, y_0, c_0), \ldots, (x_n, y_n, c_n))$ computes an output tuple $(o_0, \ldots, o_n)$, where x_i, y_i, c_i and o_i represent the core semantics' inputs/output of a semantically simple VM handler. While originally developed to minimize the number of handlers executed to improve performance, superoperators have been used by obfuscators such as Tigress primarily for obtaining more complex VM handlers.

2.2 Mixed Boolean-Arithmetic

Mixed Boolean-Arithmetic (MBA) describes an approach to encode expressions in a syntactically complex manner. The goal is to hide underlying semantics in syntactically complex constructs. First described by Zhou et al. [6], MBA algebra connects arithmetic operations (e. g., addition) with bitwise operations (e. g., logical operations or bitshifts). The resulting expressions are usually hard to simplify symbolically [39, 50], since, for every expression, an infinite number of syntactic representations exists. In general, the task of reducing MBA expressions—known as *arithmetic encodings* [12]—to equivalent but simpler expressions is NP-hard [6].

Example 5: *$f(x, y, c) := x + y$ and $g(x, y, c) := (x \oplus y) + 2 \cdot (x \land y)$ are semantically equivalent. Both implement the same core semantics, but g uses a syntactically more complex representation, called MBA.*

3 Automated Deobfuscation Attacks

In the following, we detail common techniques used to analyze obfuscated code.

Forward Taint Analysis. *Forward taint analysis* follows the data flow of so-called *taint sources*, e. g., input variables, and marks all instructions as *tainted* that directly or indirectly depend on these sources [26, 27, 29, 51]. Taint analyses are implemented with varying granularity, referring to the smallest unit they can taint. Common approaches use either bit-level or byte-level granularity. Forward taint analysis can be used to reduce obfuscated code to the instructions depending on user input. The underlying idea is that important semantics rely only on the identified taint sources. All other code constructs, e. g., as added by an obfuscator, can be omitted in an automated matter. Still, if these constructs perform calculations on the user input, taint analysis can be mislead [42, 43].

Example 6: *In Figure 1, assume `eax` is a taint source. The analysis taints the first, third, and fourth instruction since they propagate a taint source. It does* not *taint the second instruction. While its value is later used in tainted instructions, it does not directly depend on `eax`.*

Backward Slicing. Contrary to forward taint analysis, backward slicing is a backward analysis. Starting from some output variable, it recursively backtracks and marks

```
1    mov edx, eax        ; edx1 := eax1
2    mov ecx, 0x20       ; ecx1 := 0x20
3    add edx, ecx        ; edx2 := edx1 + ecx1
4    add edx, 0x10       ; edx3 := edx2 + 0x10
```

Figure 1: An assembly code snippet used to illustrate forward taint analysis, backward slicing, and symbolic execution.

all input variables on which the output depends [27, 52, 53]. In code deobfuscation, slicing can be used to find all instructions that contribute to the output. Applied to VM handlers, it allows to strip all code not directly related to a handler's core semantics. Similar to forward taint analysis, increasing the number of dependencies (e. g., by inserting junk calculations to the output) reduces the usefulness of slicing.

Example 7: *When backtracking the value of* `edx` *(line 4 in Figure 1) by following each use and definition, each instruction is marked as they all contribute to the output.*

Symbolic Execution. Symbolic execution allows to summarize assembly code algebraically. Instead of using concrete values, it tracks symbolic assignments of registers and memory in a state map [51]. Often, it works on a verbose representation of code, called *intermediate language (IL)*. Symbolic executors usually know common arithmetic identities and can perform basic simplification, e. g., constant propagation. Applied to code obfuscation, symbolic execution is used to symbolically extract the core semantics of VM handlers [40], track user input in an execution trace [27, 30, 41], or detect opaque predicates (in combination with SMT solvers) [38]. Typically, techniques to impede symbolic execution aim at artificially increasing the syntactic complexity of arithmetic operations (via MBAs) or the number of paths to analyze (triggering a so-called *path explosion*) [12, 16].

Example 8: *After symbolic execution of Figure 1, we obtain the following mappings:* `eax` *maps to itself (it has not been modified),* `ecx` *maps to* 0x20 *(line 2). The formula for* `edx` *is* `eax`+0x20+0x10*. Using arithmetic identities, the symbolic execution engine can simplify the expression to* `eax` + 0x30*.*

Program Synthesis. In contrast to other techniques that rely on syntactic analysis of obfuscated code, *program synthesis*-based approaches operate on the semantic level. They treat code as a black box and attempt to reconstruct the original code based on the observable behavior, often represented in the form of input-output samples. Approaches such as Syntia [31] and Xyntia [45] attempt to find an expression

with equivalent behavior by relying on a stochastic algorithm traversing a large search space. Other approaches, e. g., QSynth [32], are based on enumerative synthesis: they compute large lookup tables of expressions which they use to simplify parts of an expression, reducing its overall complexity. For code deobfuscation, these approaches are used to simplify syntactically complex constructs (e. g., MBAs) or to learn semantics of VM handlers. However, program synthesis struggles with finding semantically complex expressions.

Example 9: *Consider the function* $f(x, y, c) := (x \oplus y) + 2 \cdot (x \wedge y)$*. To learn* f*'s core semantics, we generate random inputs and observe* $f(2, 2, 2) = 4$*,* $f(10, 13, 10) = 23$*, and* $f(16, 3, 0) = 19$*. A synthesizer eventually produces a function* $g(x, y, c) := x+y$ *that has the same input-output behavior. Notably, it learns that parameter* c *is irrelevant.*

Ideally, superoperators provide such expressions. However, our experiments (cf. Section 6.3) demonstrate that current designs (e. g., as used by Tigress) are still vulnerable; since superoperators combine different core semantics (represented as individual inputs/output tuples), an attacker can synthesize each core semantics separately by targeting each output o_i.

Semantic Codebook Checks. A semantic codebook contains a list of expressions that an attacker expects to exist within obfuscated code. For a syntactically complex expression f, an attacker checks if f is semantically equivalent to an expression g in the codebook by using an SMT solver [54]. If the SMT solver cannot find an input *distinguishing* f and g, it *formally proved* they behave the same for all possible inputs. A typical application scenario are VM handlers: They often implement a simple core semantics (e. g., $x + y$) [21, 31]. Thus, an attacker can construct a codebook based on simple arithmetic and logical operations. As codebooks must contain the respective semantics, increasing the semantic complexity of expressions requires an (exponentially) larger codebook, making the approach infeasible for a practical application.

Example 10: *Consider a function* $f(x, y, c) := (x \oplus y) + 2 \cdot (x \wedge y)$ *and a codebook* $\text{CB} := \{x - y, x \cdot y, x + y, \dots\}$*. An attacker can consecutively pick an entry* $g(x, y, c) \in \text{CB}$ *and verify whether* $f = g$ *using an SMT solver. To this end, the solver searches an assignment that satisfies* $f(x, y, c) \neq g(x, y, c)$*. Only for* $g(x, y, c) := x + y$ *no solution can be found. Thus, the attacker proved that* f *can be reduced to a syntactically shorter expression,* $x + y$*.*

4 Design

We envision a combination of obfuscation techniques where the individual techniques harmonize and complement each other to thwart automated deobfuscation attacks. In line with this philosophy, we now present a set of generic techniques where each constitutes a defense in a particular domain. However, when these techniques are effectively combined, they exhibit comprehensive protection against automated attacks. To achieve lasting resilience, we focus on inherent weaknesses underlying existing automated attack methods, instead of targeting specific shortcomings of a given implementation. We further underline our techniques' generic nature by discussing their application on an abstract function $f(x, y, c)$ as introduced in Section 2.1. Next, we first discuss the design principles of our approach, present the attacker model, and afterwards explain the individual techniques in detail.

4.1 Design Principles

We have seen outlined automated attack methods can succeed in extracting a function f's core semantics (cf. Section 3). To mitigate these attacks, our design is based on three principles: (1) merging core semantics, (2) adding syntactic complexity, and (3) adding semantic complexity. In the following, we present techniques incorporating these principles and discuss their purpose as well as synergy effects emerging for our overall design.

Merging Core Semantics. Our first technique extends f by merging different, independent core semantics to increase the complexity. This can be understood as combining different, independent VM handlers in a single handler, or—in a more generic setting—combining different semantic operations of an unprotected unit of code in a single function f. The merge is facilitated in such a way that each core semantics is always executed. Still, as these semantics are independent of each other, we must ensure they are individually addressable, i. e., f's output is equivalent to the result of a specific core semantics. To allow the selection of the desired semantics, we extend the function definition to $f(x, y, c, k)$, where k is a *key* selecting the targeted core semantics. Formally, the selection is realized by introducing a point function $e_i(k)$, called *key encoding*, that is associated with a specific core semantics and returns 1 only for its associated key, 0 for other valid keys. This guarantees that the original semantics

are preserved. A consequence of this interlocked, “always-execute” nature is that taint analysis and backward slicing fail to remove all semantics in f not associated with a specific k.

Example 11: *We want to design a function f that returns, based on a distinguishing key, either $x+y$ or $x-y$. We write this as $f(x,y,c,k) := e_0(k)(x+y) + e_1(k)(x-y)$ where $e_i(k)$ can be any point function returning 1 for the associated k and 0 otherwise, for example $e_0(k) := k == 0xdead$. Assuming that $e_0(k)$ returns 1 and $e_1(k)$ yields 0, f returns $x+y$.*

Adding Syntactic Complexity. Assuming merged semantics using different key encodings, an attacker can still differentiate between key encoding and core semantics for a given function f, as $e_i(k)$ operates only on the key while the core semantics use x, y, and c. At the same time, a dynamic attacker with knowledge of k can employ symbolic execution to simplify f to the core semantics associated with the known k by arithmetically nullifying operations not contributing to the result. To prevent such an attack, we increase the syntactic complexity by adding Mixed Boolean-Arithmetic (MBA) formulas to key encodings as well as core semantics.

Symbolically executing these syntactically complex formulas creates no meaningful expressions. Even though modern symbolic execution engines feature simplification rules for basic arithmetic identities and laws, there exists an unlimited number of MBA representations. In general, simplifying such an expression to its syntactically smallest representative is NP-hard [6].

Example 12: *For $f(x,y,c,k) := e_0(k)(x+y)+e_1(k)(x-y)$, we can replace $x+y$ with $(x \oplus y) + 2 \cdot (x \wedge y)$, $x-y$ with $x + \neg y + 1$, and replace the multiplication of $e_1(k)(x-y)$ with the rule $(a \wedge b) \cdot (a \vee b) + (a \wedge \neg b) \cdot (\neg a \wedge b)$ for $a \cdot b$, resulting in the final function $f(x,y,c,k) := e_0(k)((x \oplus y) + 2 \cdot (x \wedge y)) + (e_1(k) \wedge (x + \neg y + 1)) \cdot (e_1(k) \vee (x + \neg y + 1)) + (e_1(k) \wedge \neg(x + \neg y + 1)) \cdot (\neg e_1(k) \wedge (x + \neg y + 1))$.*

To exploit this weakness of symbolic execution and provide a high diversity, we *synthesize* and *formally verify* MBAs instead of using hardcoded rules. This additionally complicates pattern matching and increases the number of instructions marked by forward taint analysis and backward slicing.

Adding Semantic Complexity. One of the remaining problems are semantic attacks, for example, a dynamic attacker that uses input-output behavior to learn an expression equivalent to the core semantics (e. g., via program synthesis). Therefore,

we increase the core semantics' complexity by applying the concept of superoperators (cf. Section 2.1). These superoperators make core semantics arbitrarily long and increase the search space for semantic attacks drastically.

Example 13: *Instead of using core semantics of depth 3 (e. g., $x + y$), we apply more advanced core semantics such as $(x+y)\cdot(x\oplus y))$ with depth 7 or $((x\cdot c) \ll (y \vee (x\oplus c)))$ with depth 9, resulting in $f(x, y, c, k) := e_0(k)((x + y) \cdot (x \oplus y)) + e_1(k)((x \cdot c) \ll (y \,|\, (x \oplus c)))$.*

While superoperators increase the semantic and syntactic complexity of core semantics, we further extend their syntactic complexity using MBAs. Their synergy additionally diminishes the effect of automated attacks.

4.2 Attacker Model

Intuitively, we envision a strong attacker to measure how our obfuscation scheme fares under worst-case conditions. For this purpose, we assume that an attacker has access to all automated attacks (cf. Section 3).

We assume an attacker has access to the target binary that includes at least one well-defined unit of obfuscated code at a known location. In line with our previous abstraction, we say this code unit can be represented by a function $f(x, y, c, k)$. The attacker's goal is to reconstruct the core semantics of f associated with a specific k. We require the reconstructed semantics to (1) contain *only* the core semantics associated with the specified k and (2) be comparable to the original code's semantics in terms of syntactic complexity. The intuition behind these constraints is to exclude trivial solutions such as providing the unmodified function f itself (which contains, amongst others, the core semantics for the required k).

Further, we assume two *types* of attackers, a static and a dynamic one. The *static attacker* knows the precise code locations of x, y, c, and k as well as the location of function f's output. As a result, they can enrich static analyses, e. g., by defining these code locations as taint sources. A *dynamic attacker* extends the former by the ability to inspect and modify the values at these code locations. In particular, they can observe any key k and propagate it to remove core semantics not associated with this k. While a dynamic attacker is more powerful (in terms of accessible information), certain analysis scenarios such as code running on specific hardware (e. g., embedded

devices), analysis on function-level without context, or the presence of techniques like anti-debugging [55, 56] may rule out dynamic analysis in practice.

4.3 Key Selection Diversification

We want to prevent static attackers from learning the core semantics via semantic codebook checks and prevent identification of patterns in the key selection. To do so, we employ two different key encoding schemes: Key selection based on (1) the factorization problem, and (2) synthesized partial point functions. To conduct a semantic codebook check, an attacker uses an SMT solver to check for each entry of the codebook whether it is semantically equivalent to f. Assuming that f indeed includes a matching core semantics, the SMT solver has to find a value for k such that the corresponding $e_i(k)$ evaluates to 1. One way to prevent this is to design a key encoding that relies on inherently hard problems for the SMT solver, such as factorization.

Factorization-based Key Encoding. Factorization of a semiprime n (the product of two primes, p and q) is an inherently hard problem *as long as* the size of the factors are large enough (commonly, a few thousand bits). SMT solvers prune the search space by learning partial solutions for a given problem [57], but since no partial solutions exist for factorization, they are forced to perform an exhaustive search.

We define our factorization-based key encoding as $e_i(k) := (n \mod k) \equiv 0$ where k is a valid 32-bit integer representing one of the two factors ($k \notin \{1, n\}$). As our evaluation shows, this encoding suffices to stall SMT solvers. However, its distinct structure makes pattern matching attempts easy. To increase diversity, we use MBAs and a second key encoding.

Partial Point Functions. Instead of restraining our set of key encodings to a specific type, we synthesize generic point functions without any predefined structure. This is based on the insight that the $e_i(k)$ impose only a single constraint: they must be defined for all valid keys (returning 1 for their associated one, 0 for others). Invalid keys may return arbitrary values, making our synthesized functions *partial point functions*. Consequently, we are not restricted to specific point functions, such as the factorization-based encoding, but can use arbitrary point functions fulfilling this constraint.

Given a grammar containing ten different arithmetic and logical operations (such as addition, multiplication, and logical and bitwise operations), we generate expressions by chaining a randomly selected operation with random operands. This operand is either

an arbitrary key byte or a random 64-bit constant. We chain at most 15 operations to limit the overhead resulting from this expression. Finally, we check if the synthesized expression satisfies the point function's constraint.

Example 14: *Let* $(k_0, k_1, k_2) := (\text{0x1336}, \ \text{0xabcd}, \ \text{0x11cd})$ *be a set of keys. Then, we synthesize the point function* $e_0(k) := ((\text{0xff} \wedge k) \oplus \text{0xcd}) \cdot \text{0x28cbfbeb9a020a33}$ *for a 64-bit vector* k*.* $e_0(k)$ *evaluates to* 1 *for* k_0 *and to* 0 *for* k_1 *and* k_2*. For all other keys, it returns arbitrary values.*

4.4 Syntactic Complexity: MBA Synthesis

To thwart symbolic execution and pattern matching, we use MBAs for all components, including core semantics and key encodings. As hardcoded rules only provide low diversity, we precompute large classes of semantically equivalent arithmetic expressions and combine them through recursive, randomized expression rewriting. We now detail the creation of the equivalence classes and discuss our term rewriting.

Equivalence Class Synthesis. To create semantic equivalence classes for expressions, we rely on enumerative program synthesis [58, 59]. To this end, we first define a context-free grammar with a single non-terminal symbol S as start symbol and two terminal symbols, x and y, representing variables. For each arithmetic operation, we define a production rule that maps the non-terminal symbol to arithmetic operations (e. g., addition) or terminal symbols. To apply a specified production rule to a non-terminal expression, we replace the left-most S with the rule. Expressions without a non-terminal symbol can be evaluated by assigning concrete values to x and y. We say that the *depth* of an expression represents the number of times a non-terminal symbol was replaced by a production rule.

Example 15: *The grammar* $(\{S\}, \Sigma = V \cup O, P, S)$ *with the variables* $V = \{x, y\}$*, the set of arithmetic symbols* $O = \{+, -\}$ *and the production rules* $P = \{S \to x \mid y \mid (S + S) \mid (S - S)\}$ *defines the syntax of how to generate terminal expressions. To derive the expression* $x + y$ *of depth* 3*, we apply the following rules:* $S \to (S+S) \to (x+S) \to (x+y)$*. With a mapping of* $\{x \mapsto 2, y \mapsto 6\}$*, we can evaluate the terminal expression to* 8*.*

We now describe how we use our context-free grammar in combination with Algorithm 1, which illustrates the high-level approach of equivalence class synthesis. Starting with a worklist of non-terminal states (initialized with the start symbol S), we itera-

Algorithm 1: Computing equivalence classes.

```
Data: n is the maximum depth.
1 states ← {S}
2 for d ← 1 to n do
3     terminals ← derive_terminals(states)
4     process_terminals(terminals)
5     non_terminals ← derive_non_terminals(states)
6     states ← non_terminals
```

tively process all expressions for a certain depth until we reach a specified upper bound depth N. For a given depth, we derive all terminal and non-terminal expressions (also referred to as *states*) before processing the terminals and then repeating the process for the next depth. The call to `process_terminals` is responsible for sorting the expressions into the respective equivalence classes. To this end, we evaluate all expressions for a high number of different inputs (e. g., $1,000$), recording their output. Expressions with the same output behavior for all provided inputs are sorted into the same equivalence class. This provides an effective but coarse-grained sorting of expressions into potential equivalence classes. In a final step, we verify that these classes are semantically correct. For this, we choose the member with the smallest depth as representative and check with an SMT solver that all other members are semantically equivalent to this representative. Expressions failing this check are removed from the equivalence class. All remaining expressions are formally proven to not alter the original semantics.

To prune the search space and avoid trivial expressions (e. g., $x+0$), we symbolically simplify each terminal and non-terminal expression. For this purpose, we apply a normalization step to commutative operators, perform constant propagation, and simplify based on common arithmetic identities (e. g., $x+y-y$ becomes x).

Expression Rewriting. So far, we generated a large set of diverse equivalence classes we can use for replacing syntactically simple expressions with more complex ones. A naive approach replacing expressions with MBAs from the equivalence classes is bounded by the largest depth found in the respective class. To overcome this limitation, we propose a recursive expression rewriting approach using the equivalence classes as building blocks. This allows us to create expressions of *arbitrary* syntactical depth. Even assuming an attacker is in possession of all rewriting rules, it is difficult to invert an expression: Term rewriting is inherently *destructive* [60]. Without knowing the applied

rewriting rules and their order, an attacker has to check all possibilities: n rewriting rules applied over m layers, resulting in the prohibitively large number of n^m candidates.

Given some expression e, we pick a random subexpression and check if it is a member of an equivalence class. If it is, we randomly choose another member from this class and use it to replace the picked subexpression within e. We recursively repeat this process for a randomly determined upper bound n. As all members within an equivalence class are proven to be pairwise equivalent, each replacement is guaranteed to produce an equivalent expression. Consequently, the final expression is provably equivalent to the first.

Example 16: *Assume that we want to increase the syntactic complexity of* $e := (x + y) + z$ *with the upper bound* $n = 2$*. First, we randomly choose the subexpression* $x + y$*. We then pick another member of the same equivalence class—*$(x \oplus y) + 2 \cdot (x \wedge y)$*—and replace it in* e*. In this case, we obtain* $e := ((x \oplus y) + 2 \cdot (x \wedge y)) + z$*. In a second step, we choose* $x \oplus y$*, pick the semantically equivalent member* $(x \vee y) - (x \wedge y)$ *and replace it again. The final MBA-obfuscated expression is* $e := (((x \vee y) - (x \wedge y)) + 2 \cdot (x \wedge y)) + z$*.*

Empirical testing showed that for an initial expression the randomly picked subexpressions would often be short, causing the resulting recursive rewriting to be very local in nature rather than considering all of the expression. The previous example illustrates this behavior. Considering the expression as an *abstract syntax tree (AST)*, we twice replaced deeper parts of the AST while ignoring the top-level operation (addition with z). Consequently, subsequent iterations would be even less likely to pick the high-level operation, considering the wealth of other operations to pick from. Therefore, the AST would be significantly unbalanced. To avoid this, we prefer selecting top-level operations in the first loop iterations.

4.5 Semantic Complexity: Superoperators

Up to this point, f's core semantics have a rather low semantic complexity (e. g., $x + y$). To thwart semantic attacks, we use a variation of superoperators that increase the semantic complexity. The intention is to significantly increase the search space for an attacker: For example, assume a set of three variables V and a set of six binary operations O: For semantic depth 3 (e. g., $x+y$), an expression contains $m = 2$ variables and $n = 1$ operations, such that an attacker has to brute-force at most $|V|^m * |O|^n = 3^2 * 6^1 = 54$ possibilities. For depth 7 (e. g., $((x + y) \cdot (x \oplus c))$), they must try up to

$3^4 + 6^3 = 17,496$ different expressions (or $314,928$ for depth 9). In other words, the search space grows exponentially, making semantic code book checks as well as program synthesis infeasible.

However, common superoperator strategies, e. g., as used by Tigress [61], are not resilient against these attacks (cf. Section 6.3). They usually include independent core semantics, each having their own output; this causes the handler to have multiple, independent outputs, which an attacker can target individually. As each core semantics itself usually implements only a single operation [21, 31] (e. g., $x+y$ with semantic depth 3) attacking one such superoperator is similar to attacking a series of regular handlers. To avoid this pitfall, we design our superoperators to preserve the signature of f (a *single* output and x, y, c and k as inputs) while providing a high semantic depth. In other words, our superoperators consist of a chain of core semantics that depend on each other and must be executed sequentially: The output of the core semantics is used as input for subsequent core semantics; the last core semantics produces the output of the handler. Even if an attacker is aware of these superoperators, they cannot split a handler into multiple separate synthesis tasks and forces them to synthesize the whole expression.

On a technical level, we construct superoperators based on data-flow dependencies, more precisely *use-definition chains* based on *static single assignment (SSA)* [62]: Given an unprotected code unit in form of instructions in three-address code, we assign a unique variable to each variable definition and replace subsequent variable uses with its latest definition on the right-hand side (called *SSA form*). Then, we build superoperators by first randomly picking variables on the right-hand side and then replacing these *uses* by their respective variable *definitions* recursively. By choosing lower and upper limits for the recursion bound, we can control the superoperators' semantic depth. To further increase the syntactic complexity, we apply our MBAs.

Example 17: *Assume we have three sequential instructions (Figure 2, l. 1-3) implementing semantically simple operations; each represents an individual core semantics. Notably, the first instruction's output serves as input for the second and third. Similarly, the second instruction is an input to the third. To create a superoperator that implements a semantically more complex operation, we transform the code into SSA form, (randomly) pick* `b1` *in the third instruction and replace this use by its definition (l. 2), yielding* `d2 := (a * d1) | d1`*. When picking* `d1`*, we replace it by its definition*

```
1 | d := a + b        ;  d1 := a  + b
2 | b := a * d        ;  b1 := a  * d1
3 | d := b | d        ;  d2 := b1 | d1
```

Figure 2: Three different core semantics, each implementing a simple operation. On the right-hand side, the SSA form of the respective expressions.

(l. 1) accordingly, transforming the expression into `d2 := (a * (a + b)) | (a + b)`*. While the initial expressions have a semantic depth of 3, the superoperator's depth is 9.*

Intuitively, replacing a *use* by its respective *definition* is guaranteed to preserve the semantics, as variable assignments are immutable in SSA form. Additionally, we prove the rewritten superoperator is equivalent to the original code with symbolic execution.

4.6 Synergy Effects

To summarize, each of our components thwarts specific deobfuscation attacks: MBAs tackle symbolic execution and pattern matching, while the nature of f with its multiple core semantics, selected via a key, prevents taint analysis and backward slicing from removing irrelevant semantics. Further, superoperators increase the semantic complexity, throwing off semantic attacks.

As indicated, especially the combination of our techniques prevents automated deobfuscation attacks: They do not only co-exist but have beneficial synergy effects, which in turn improve the overall resilience of the combination. For example, our MBAs weaken pattern matching on all levels, including key encodings, and cause the differences between key encoding and core semantics to blur. Besides the syntactic confusion introduced, we can propagate the core semantics into the key encoding and vice versa. For instance, we may use MBAs that extend the key check with the variables x or y using arithmetic identities that do not alter the key check itself. At the same time, MBAs benefit from superoperators given they provide ample opportunity to pick and replace subexpressions.

4.7 Verification of Code Transformations

Obfuscation generally modifies the syntactic representation of code; thus, it is crucial to verify that it does not change the code's semantic behavior. One can achieve this by

checking if the transformed code is semantically equivalent to the original one. While this works well for short sequences of instructions (e. g., by using SMT solvers) within a reasonable amount of time, it does not scale to complex programs. In such cases, the industrial state of the art approximates these guarantees by using extensive random testing [63, 64].

For our design, we choose the best applicable verification method to ensure correctness: For individual components, we use formal verification to prove their correctness (cf. Sections 4.3, 4.4, 4.5). To improve the confidence of the correctness of the combination, we use an approach similar to black-box fuzzing [65, 66], where we compare the I/O behavior of the original and transformed code for a user-configurable number of random inputs, usually ranging from $1,000$ to $10,000$. These are randomly sampled depending on the type expected by the program (e. g., ASCII strings, random 64-bit integers, or known edge cases such as 0 or $0xff..ff$), which needs to be specified by the user. Crucially, we rely on human insight and careful specification of the input domain such that the sampled inputs cover the full program functionality. We apply this fuzzing both on the binary level as well as on the intermediate representation; for the former, we compare the compiled versions of the unprotected and protected programs, while we emulate the program's intermediate representation before and after transformations for the latter. As a consequence of our handlers' interlocked, always-execute nature, we achieve full code coverage and path coverage both on the intermediate representation as well as on the binary level for all handlers needed to represent the original code.

5 Implementation

To evaluate our techniques, we implement a VM-based obfuscation scheme named Loki on top of LLVM [67] (version 9.0.0) and a code transformation component written in Rust. Loki consists of ~3,100 LOC in C++ and ~8,700 LOC in Rust. In this scheme, each function $f(x, y, c, k)$ is represented by one of our 510 handlers. In other words, each handler can implement any semantic operation that requires no more than two input variables and one constant. Our handlers support the inclusion of three to five core semantics (randomly chosen at creation time), which can be addressed by setting k accordingly. Besides these 510 handlers, we have a *VM exit* and a handler managing memory operations. The control flow between handlers is realized as *direct threaded code* [68], i. e., each handler inlines the VM dispatcher. Our VM assumes a 64-bit

architecture. Code operating on smaller bit sizes is semantically upcasted to guarantee correctness.

Our approach to obfuscate real-world code consists of three major steps: Lifting, code transformation, and compilation. The *lifting* starts with a given C/C++ input program that contains a specified function to protect. We then translate this function to LLVM's intermediate representation (IR) and use various compiler passes to optimize the input and unroll loops as our prototype does not support control-flow to reduce engineering burden. Note that this is no inherent limitation of our approach, but a simplification we made as LLVM's passes sufficed in creating binaries that our prototype implementation can process. Finally, we lift the resulting LLVM IR to a custom IR which the code transformation component internally operates on. This component (a) parses the lifted representation of the targeted function, (b) creates superoperators based on this input (with recursion bound 3 to 12), (c) instantiates the VM handlers, applies our obfuscation techniques (e. g., MBAs), and verifies them. For MBAs, we use a random recursive expression rewriting bound between 20 and 30. We choose from a pre-computed database of 843, 467 MBAs (all expressions up to a depth of 9), split over 48 equivalence classes. In each class, there are roughly 17,500 entries on average. To exemplify the dimensions: An attacker has to try up to $n^m_{Loki} = 843,467^{30} = 6.1 * 10^{177}$ possibilities to simplify our MBAs; Based on our reverse engineering efforts, state-of-the-art obfuscator Tigress features only 47 hand-crafted rules (that are not applied recursively), such that an attacker has to evaluate $n^m_{Tig} = 47^1 = 47$ possibilities. (d) Finally, the Rust component generates the VM bytecode and translates the handlers back into LLVM IR. Then, obfuscated and original code are compiled with `-O3` and verified.

6 Experimental Evaluation

Based on our prototype implementation, we evaluate if our approach can withstand automated deobfuscation techniques (*resilience*), while maintaining *correctness* and imposing only acceptable overhead (*execution cost*). Overall, we follow the evaluation principles outlined by Collberg et al. [69].

All experiments were performed using Intel Xeon Gold 6230R CPUs at 2.10 GHz with 52 cores and 188 GiB RAM, running Ubuntu. Our obfuscation tooling uses LLVM [67] (v. 9.0) and the SMT solver Z3 [70] (v. 4.8.7). For tracing coverage, we rely on Intel Pin [71] (v. 3.23). Our deobfuscation tooling is based on Miasm [72] (commit `65ab7b8`),

Triton [73] (v. 0.8.1), and Syntia [31] (commit `e26d9f5`). We use our prototype of Loki and the academic state-of-the-art obfuscator, Tigress [61] (v. 3.1), to obfuscate five different programs, each implementing a cryptographic algorithm: AES, DES, MD5, RC4, and SHA1. This is a common approach: the first three are based on an obfuscation data set provided by Ollivier et al. [16]; the others are adapted from reference implementations [74, 75]. These algorithms are representative for real-world scenarios in which cryptographic algorithms are used to guard intellectual property (e. g., hash functions used for checksums in commercial DRM systems) [4]. In a case study, we obfuscate VLC's DVD decryption routine to show how Loki can be applied onto real-world use cases. Where necessary, we adapt the programs slightly to allow Loki to process them (cf. Section 5) without modifying their functionality. Tigress' configuration [48] resembles our design and works on the same source code files.

6.1 Benchmarking

Our goal is to benchmark the *correctness* and *cost* of our obfuscator. We do so by conducting a series of experiments, measuring the overhead in terms of runtime and disk size as well as verifying the correctness of transformed code. For each obfuscator, we create $1,000$ obfuscated instances for each of the five targeted programs and use them for all experiments. The overhead comparison is given as factor relative to the original, unobfuscated program compiled with `-O3`. To measure the MBA overhead, we create another $1,000$ obfuscated instances without any MBAs for Loki.

Experiment 1: Correctness. *For each target, we verify that all* $1,000$ *obfuscated instances produce the same output as the original program for more than* $1,000,000$ *inputs. To obtain a uniform distribution over varying input lengths of our cryptographic targets, we create* $10,000$ *random inputs for each supported input length* $l \in [16;128]$*. Additionally, we test a number of edge cases* $\in \{0x0...0, 0xff...ff, 0x80...00, 0x00...01, 0xaa...aa, 0x55...55\}$ *(or their cartesian product if two inputs are required). This amounts to a total of* $1,134,068$ *inputs, for which we assert equal input-output behavior.*

All obfuscated binaries (both those with and without MBAs) exhibit exactly the same behavior for the $1,134,068$ inputs tested.

Experiment 2: Code Coverage and Path Coverage. *To further increase confidence in our correctness tests, we measure both the code coverage and the path coverage*

that the inputs from Experiment 1 achieve on the to-be-protected code both for the original program and obfuscated instances.

We find that each of the more than 1,000,000 inputs from Experiment 1 achieves full code coverage and full path coverage. This ensures that our inputs cover the complete behavior of the code we obfuscate and that our obfuscation transformations have not altered this behavior.

Experiment 3: Overhead. *For each target, we measure the average execution runtime. To this end, each target wraps the to-be-protected code in a single function, which is called* 10,000 *times per input. We then execute each obfuscated binary for* 1,000 *random inputs, recording the collected timings. We also compare the original program's disk size to the average of the obfuscated binaries.*

As evident from Table 2, the runtime overhead ranges from a factor of 301 to 482 compared to the original program's execution time. While this overhead may appear excessive—also in comparison to Tigress—state-of-the-art commercial obfuscation generally imposes an even larger slowdown, up to ten times more than Loki (cf. Table 2, [76]). We re-run this experiment on the 1,000 binaries without MBAs to evaluate their impact. On average, they are responsible for ~39% of the overhead. Similar for the disk size, the obfuscated programs are 18 to 51 times larger than the original ones. Size-wise, MBAs cause ~33% of the overhead. For further details of our MBAs' overhead, we refer to the Technical Report [48]. Compared to Themida and VMProtect, our obfuscating transformations generate almost always smaller programs, while Tigress always produces significantly smaller binaries.

Overall, we conclude that our overhead is moderate in comparison to commercial state-of-the-art obfuscators. For further discussion, we refer to Section 7. Tigress' overhead is impressively small, but it falls short in providing comprehensive protection as the following experiments show.

Case Study: VLC with libdvdcss. To showcase the practical feasibility of Loki in real-world scenarios, we obfuscate the `DecryptKey` function in libdvdcss [77]; this component of VLC [78] is responsible for decrypting the multimedia content of DVDs keys. The underlying idea is to protect the decryption algorithm from the prying eyes of crackers and protect intellectual property. However, the vast majority VLC's code is irrelevant to content decryption, such that there is no need to obfuscate the whole libdvdcss library or even the whole media player. After obfuscating the `DecryptKey`

Table 2: Runtime and disk size overhead as factors relative to the non-obfuscated binaries (compiled with `O3`). *(w/o = without)*

		Time Factor					Size Factor				
		AES	DES	MD5	RC4	SHA1	AES	DES	MD5	RC4	SHA1
VMProtect	Virtualization	2,489	1,859	1,982	1,321	2,524	37	21	40	44	40
	Ultra	8,925	9,152	13,047	5,806	15,411	47	37	57	59	53
Themida	Tiger White	1,388	622	203	240	552	58	38	58	58	59
	Dolphin Black	11,695	5,052	2,428	3,634	8,354	67	47	85	63	84
Loki		386	301	357	482	386	33	18	39	37	51
	w/o MBA	236	185	204	315	233	21	13	25	26	32
Tigress		261	51	101	58	111	3	4	2	2	3

function, which is called *before* the actual media content is played, we measure the execution time of the function during initial startup, when the DVD is decrypted. We average the results over ten executions. We find that without obfuscation, the function is executed in $2,952$ nanoseconds, while with obfuscation, the decryption lasts $937,606$ nanoseconds. Overall, Loki slows down the initialization by one millisecond, a negligible cost for protecting one's intellectual property, especially if the to-be-protected function is only called in the application's startup phase.

6.2 Resilience

We evaluate whether our techniques can withstand automated deobfuscation approaches. To this end, we analyze the impact of syntactic and semantic attacks against the obfuscated code in the presence of both static and dynamic attackers. We design all experiments by assuming the strongest attacker model. To this end, we test each component individually, therefore ignoring beneficial synergy effects. Where applicable, we first evaluate our techniques on a general design level before testing their concrete implementations. The former serves as universal evaluation of a technique's resilience, while the latter demonstrates that this also holds when actually implemented on the binary level.

LokiAttack. Fundamentally, attacking the obfuscated VM on the binary-level has two stages: (1) Identifying a specific handler within the VM, and (2) attacking (simplifying) this particular handler as far as possible. For our evaluation, especially (2) is interesting, as all our techniques focus on hardening individual handlers. As such, we

develop a custom attack framework that we call LokiAttack. It is specifically tailored to the attacked obfuscators and automates the first stage: It identifies all VM handlers and provides the attacker (for each handler) with access to the handler parameters (x, y, c, and—for Loki—k). For a dynamic attacker, it also provides concrete values for these parameters. Finally, LokiAttack uses symbolic execution to obtain all code paths through the intermediate representation (IR) of the `O3`-optimized VM code that depend on an *unknown* (static attacker) or *known* (dynamic attacker) value of k (for Loki). For each such path, an attacker can launch the actual attack on the handler (stage 2), for which LokiAttack provides a number of techniques implemented as plugins, e. g., taint analysis, symbolic execution, or program synthesis. To implement LokiAttack, we use Miasm; the plugins for stage 2 are based on Triton (byte-level taint analysis), Miasm (bit-level taint analysis, backward slicing, and symbolic execution), and Syntia (program synthesis). These plugins include costly operations (SMT solving, program synthesis, and symbolic execution), from which some may run for several days. As our evaluation consists of more than $300,000$ analysis tasks, we limit each one to 1 hour to keep the analysis time manageable. This is a common use-case and in-line with previous work on deobfuscation [31, 38, 45].

6.3 Evaluation of Key Encodings

We evaluate whether a static attacker can obtain a specific core semantics using semantic codebook checks. Note this experiment is only applicable to Loki as Tigress has no concept of key-based selection of core semantics. Assume that the function $f(x, y, c, k)$ includes $x + y$ as one of its core semantics. Then, an attacker can use an SMT solver to find a value for k such that f is semantically equivalent to $g(x, y, c) := x + y$. On a technical level, we employ an approach called *Counterexample-Guided Abstraction Refinement (CEGAR)* [79, 80] that relies on two independent SMT solvers: While SMT solver A tries to find assignments for all variables (including k) such that f and g produce the same output, solver B tries to find a counterexample for this value of k such that f and g behave differently. Then, A uses the counterexample as guidance.

Experiment 4: Hardness of Key Encodings. *We generate* $1,000$ *random instances of our factorization-based key encoding and synthesize* $10,000$ *point functions. Then, we apply the CEGAR approach independently to both key encodings and check if the SMT solver finds a correct value for* k*.*

We observe that the SMT solver found no correct key for the factorization-based encoding, but hit the 1h timeout in all cases. Considering the point function-based key encoding, Z3 managed to find a value for k in $6,932$ cases (~69%). On average, it found the solution in 284s (excluding timeouts). We conclude that an SMT solver struggles with our factorization-based key encoding, while point functions often can be solved. Recall though that point functions primarily serve to diversify and erase discernible patterns to impede pattern matching.

Experiment 5: Key Encoding on Binary Level. *To verify if our implementation properly emits these key encodings, we generate* $1,000$ *binaries that contain one specific handler which includes* $x+y$ *as one of its core semantics. These binaries contain neither MBAs nor superoperators. Assuming a static attacker uses CEGAR, we check in how many cases the SMT solver finds a correct value for* k.

Using LokiAttack, we obtain the handler's instructions and use our CEGAR plugin based on Z3 to find a value for k, such that these instructions are semantically equivalent to $x + y$. While hitting the timeout in 690 cases, Z3 managed to find a correct value for k in 310 cases (31%). The SMT solver needed, on average, 444s to find the solution (excluding timeouts). Overall, we conclude that our key encodings indeed pose a challenge for a static attacker relying on SMT solvers.

Note that this component is special within our system, as its approach specifically targets only static attackers. This is due to the fact that dynamic attackers can trivially observe a value for k. While a dynamic scenario is not always possible, another attack vector could be to offload 64-bit integer factorization to custom tools (assuming an attacker manages to locate the key encodings, which in itself is a non-trivial task given our MBAs and point functions). Thus, our key encodings can be considered to be our weakest component. However, our design assumes that an attacker can retrieve a value for k, but we try to make this as hard as possible. The syntactic simplification experiments show that knowledge of a key k is beneficial but not sufficient to simplify any handler.

Syntactic Simplification. In the following, we evaluate whether syntactic simplification techniques—namely, forward taint analysis, backward slicing, and symbolic execution—succeed in extracting a core semantics associated with a specific key, either by trying to identify instructions not contributing to a function f's output or by symbolically simplifying f. We use LokiAttack as a basis and conduct the re-

Table 3: Statistics for backward slicing and forward taint analysis (TA), averaged over 7,000 handlers. Unmarked instruction can be removed as irrelevant. *(Unmark. = not tainted / not sliced; Dyn. = dynamic attacker)*

		Byte-level TA		Bit-level TA		Slicing	
		Static	Dyn.	Static	Dyn.	Static	Dyn.
Loki	IR paths	1,950	199	1,451	168	1,656	179
	Unmark.	17.49%	17.50%	17.61%	17.62%	5.49%	7.57%
	Time [s]	556	58	710	78	630	67
Tigress	IR paths	1		1		1	
	Unmark.	44.70%		44.70%		22.35%	
	Time [s]	1.3		1.6		1.4	

spective attack in stage 2 for both a static and a dynamic attacker. We assume that an attacker is given a binary containing seven handlers, $f_0(x, y, c, k), \cdots, f_6(x, y, c, k))$, each containing between 3 and 5 core semantics. Further, each handler f_i contains one predefined core semantics from the set $\{x + y, x - y, x \cdot y, x \wedge y, x \vee y, x \oplus y, x \ll y\}$ that an attacker wants to identify via syntactic simplification. As sample set for our experiments, we generate 1,000 binaries protected by MBAs but without superoperators, amounting to 7,000 handlers to analyze. For each binary, we use LokiAttack to extract all handlers; for each handler, LokiAttack provides us with the parameter locations (and values for the dynamic scenario) and all code paths. For each code path (a list of instructions), we then use the respective stage 2 plugin. We apply the following experiments also to 7,000 Tigress handlers (with disabled superoperators), respectively.

Experiment 6: Forward Taint Analysis. *For each of the* 7,000 *handlers, we conduct a forward taint analysis with byte-level and bit-level granularity. The former is based on* Triton*, while the more precise bit-level taint analysis is implemented on top of* Miasm*. In general, higher precision is expected to produce fewer false positives and result in fewer tainted instructions. Recall that an attacker's goal is to identify all instructions that do not belong to the core semantics associated with a specific key. Using*

taint analysis, an attacker can remove all instructions not depending on x, y, c, or k (in a dynamic scenario: on a concrete *value for k).*

The resulting data is shown in Table 3 (where *unmarked* instructions refer to instructions that are *not* tainted, i. e., instructions that can be removed). The results show two interesting insights: First, the granularity has negligible impact on the results (difference of 0.12%). Second, the number of tainted instructions is almost equal for a static and a dynamic attacker. This is surprising as for Loki the number of visited paths in the IR's control-flow graph is significantly lower in the dynamic setting. Intuitively, this means a dynamic attacker has better chances of removing more instructions. However, our results show that the sole benefit of a dynamic attacker is spending less time per handler. In numbers, an attacker is always able to only remove about ~18% of a handler's assembly instructions. Manually inspecting the instructions not tainted revealed that these can always be put into two categories: Either they are part of the inlined VM dispatcher that is responsible for loading the next handler (which is independent of the current handler's semantics), or it is an instruction loading a constant value before it interacts with tainted instructions (comparable to Example 6). To summarize, forward taint analysis fails to remove a single instruction that is related to the core semantics or key encodings. For Tigress, on the contrary, only one IR path exists, meaning the handlers are short and simplistic in nature. No difference between bit and byte-wise taint analysis exists; overall, an attacker succeeds in removing 45% of instructions—significantly more than for Loki.

Experiment 7: Backward Slicing. *Besides forward taint analysis, an attacker can use backward slicing to identify all instructions that contribute to a handler's output. We again consider both a static and dynamic attacker trying to reduce each handler to the core semantics associated with a specific k by removing as many unrelated instructions as possible. Our backward slicing approach is based on* Miasm *and operates on the same* $7,000$ *handlers as Experiment 6.*

The results are denoted in Table 3, where an *unmarked* instruction refers to an instruction that was not sliced, i. e., it does not contribute to the output. Other than for taint analysis, a dynamic attacker has a slight advantage compared to a static attacker (2.08%), as they slice slightly fewer instructions. While the static attacker marks all instructions but the inlined dispatcher, our manual inspection revealed that dynamic analysis skips some IR paths depending on irrelevant key values. Compared

Table 4: Symbolic execution for semantic depth 3 and 5, each averaged over 7,000 handlers. *(Simplified = percentage of handlers simplified)*

		Depth 3		Depth 5	
		Static	Dynamic	Static	Dynamic
Loki	IR paths	4,960	559	5,450	703
	Simplified	0%	17.93%	0%	14.64%
	Time [s]	–	94	–	168
Tigress	IR paths	1		–	
	Simplified	30.61%		–	
	Time [s]	1.4		–	

to forward taint analysis, backward slicing marks more instructions, i. e., it removes fewer instructions (~7% vs. ~18%). This is due to the backward-directed nature of the approach, which allows it to slice instructions loading constant values. We conclude that backward slicing is technically more precise than forward taint analysis, but fails to remove instructions belonging to the core semantics or key encodings. For Tigress, slicing succeeds to remove significantly more instructions, however, less than taint analysis. This is again due to the loading of constant values.

Experiment 8: Symbolic Execution. *Besides removing instructions not contributing to the output, an attacker can use symbolic execution to extract a handler's core semantics. To this end, a symbolic executor uses simplification rules to syntactically simplify the handler's semantics as much as possible. We use the same* 7,000 *handlers as Experiment 6. We analyze each handler independently with* Miasm*'s symbolic execution engine and measure whether it can be simplified to the original core semantics.*

We model both a static and more powerful dynamic attacker. In the latter scenario, the attacker observes a value for k and thus can nullify all core semantics not related to this specific k. Hence, they obtain a much simpler expression containing only the desired core semantics, albeit in syntactically complex form (due to MBAs). Recall that for the 7,000 handlers, the semantic depth of the core semantics is always 3 (e. g., $x + y$). This intentionally weakens resilience, as superoperators with a higher depth naturally increase both semantic and syntactic complexity. To show this, we create another 7,000 handlers (1,000 binaries à 7 core semantics) with a semantic depth of 5

(e. g., $x + y + c$) and repeat this experiment. We cannot create handlers of depth 5 for Tigress, as it is not possible to explicitly set the handler's semantic depth.

All results are shown in Table 4. Notably, a static attacker fails to simplify any of Loki's handlers. Without knowing a value for k, an attacker has to analyze the expression containing *all* key encodings and their associated core semantics. In other words, an attacker fails to nullify irrelevant core semantics. To significantly simplify the handler, a static attacker has to find a valid key first (reducing the problem to Experiment 5). A dynamic attacker, on the other hand, only has to simplify the MBAs as they already identified the core semantics associated with the key. For depth 3, they succeed in removing all MBAs for ~18% of Loki's handlers, while, for Tigress, ~31% of the handlers can be simplified. In other words, an attacker can simplify significantly more handler for Tigress than for Loki. For a more realistic depth of 5—subsequent experiments show ~80% of Loki's handlers are at least of depth 5—the attacker's success rate is even lower, namely ~15%. This percentage implies that a number of expressions can be simplified regardless of the higher base depth. This may be the case, e. g., when the random combination of applied MBAs cancels itself out. Still, this demonstrates our synergy effects are indeed helpful to prevent an attacker from symbolically simplifying the core semantics, leaving them with a complex MBA that conceals the actual semantics.

We conclude that our MBAs are successful in thwarting symbolic execution, one of the most powerful deobfuscation attacks. For a more detailed analysis of how a user of Loki can trade performance against reducing the attacker's success chances even further (to 6.79%), refer to the Technical Report [48].

Experiment 9: Diversity of MBAs. *An attacker tasked with removing such MBAs may investigate whether a diverse number of expressions exists for the same core semantics. If this is not the case, they can manually analyze each MBA and extend the symbolic executor's limited set of simplification rules by rewriting rules to "undo" specific MBAs. To this end, we assume a dynamic attacker that already symbolically simplified the expression as far as possible without any MBA-specific simplification rules. We do this for each handler type (recall that the* $7,000$ *handlers of depth 3 consist of* 7 *different core semantics à* $1,000$ *handlers) and then analyze how many different, unique MBA expressions exist.*

Our analysis reveals that, in summary, Loki generates 5,482 unique MBAs for the 7,000 expressions analyzed (78.31%), while Tigress creates only 16 (~0.23%) unique MBAs. Thus, an attacker adding 16 rules to their symbolic executor could simplify all core semantics. This difference can be explained by the fact that Tigress uses only a few handwritten rules to create MBAs, while Loki features a generic approach to synthesize MBAs. To further highlight the difference between both approaches, we repeat this experiment for another set of 7,000 handlers—created in the same configuration but with different random seeds—and calculate the intersection of unique MBAs. Tigress re-uses exactly the same 16 MBAs, while Loki re-uses 109 expressions but generates 5,299 new unique MBAs (i. e., 10,781 unique MBAs in total). Creating simplification rules specific to Loki is a tedious task (given the high number of unique MBAs) that does not pay off when analyzing other obfuscated instances. For a discussion of what an attacker can achieve when they are in possession of *all* available MBA rewriting rules, refer to Section 7. We conclude that Loki's MBAs are superior to state-of-the-art approaches relying on a small number of hardcoded MBAs, both in terms of resilience and diversity.

State-of-the-art MBA Deobfuscation. A number of approaches for MBA simplification have been proposed, most notably SSPAM [36], arybo [37], NeuReduce [46], and MBA-Blast [47]. The deployed techniques range from pattern matching-based simplification over machine learning to algebraic simplification. Regardless of the underlying technique, they all share one major drawback: They expect the MBAs to be available on the source code level in form of a formula, such as "x + y - y", rather than dealing with them on the binary level. As a consequence, these deobfuscation tools lack support for MBAs using different bit sizes and operations such as zero-extension or sign-extension. Furthermore, they assume that the MBAs are free of constants and more complex arithmetic operations, such as multiplication or left-shifts. In contrast to these limitations and assumptions, Loki's MBAs not only employ all these operators but also contain constants, thus making a fair, direct evaluation of our MBAs contained in binaries difficult. To avoid these pitfalls, we use Loki's term rewriter to generate simpler MBAs—called *artificial* MBAs—and emit them as a formula rather than deploying them in the binary. We make the following artificial restrictions: We (a) emit no constants, (b) do not intertwine the MBAs with the key encoding, (c) remove any information (or operation) relating to size casts, and (d) avoid complex, unsupported operations such as multiplications or left-shifts. Instead, the resulting MBAs are a for-

mula containing only the following operations $\{+, -, \wedge, \vee, \oplus\}$. While this significantly weakens Loki's MBAs, aforementioned state-of-the-art MBA deobfuscation techniques can now process these artificial MBAs, allowing a fair evaluation. For NeuReduce, we use the Gated Recurrent Unit (GRU)-based Long Short-Term Memory (LSTM) model provided by the authors. We further adapt MBA-Blast to recursively attempt simplification for subexpressions. MBA-Blast cannot deal with nested arithmetic expressions; only expressions on the root level of the expression's abstract syntax tree may contain arithmetic operators. All subexpressions must consist purely of Boolean operators. As our MBAs are highly nested, we apply the respective tool recursively on each subexpression until it cannot simplify the expression any longer. We considered evaluating arybo [37]; however, we noticed it does not terminate for 64-bit expressions within one hour. Further, arybo outputs truth tables in form of expressions representing the relations between different bit positions. Its goal is aiding a human analyst rather than automated simplification. Thus, we exclude it from the following experiment. As a baseline, we port our deobfuscation tooling, LokiAttack with the SE plugin, to the source level: We first use aggressive compiler optimizations ("`-O3`") to simplify the MBA and then—as a stage 2 plugin—symbolically summarize it using Miasm's symbolic execution engine. This is the same approach as has been used for the previous experiments.

Experiment 10: MBA Formula Deobfuscation. *For each core semantics from the set* $\{x + y, x - y, x \wedge y, x \vee y, x \oplus y\}$*, we use* Loki *to generate* $1,000$ *artificially simplified MBAs on the source code level. We do this for each recursive term rewriting bound from* $[1, 30]$ *(during normal operation,* Loki*'s rewriting bound is randomly chosen between* $[20, 30]$*). In summary, we generate* $5,000$ *MBAs per rewriting bound, i. e.,* $150,000$ *obfuscated expressions in total. We then pass each MBA to the deobfuscation tools* MBA-Blast*,* NeuReduce*,* SSPAM*, and* LokiAttack *and observe how many expressions they can simplify to the ground truth.*

The number of simplified expressions, averaged over the five different core semantics, are depicted in Figure 3. As the data shows, our custom deobfuscation tooling, LokiAttack, significantly outperforms all state-of-the-art deobfuscation techniques. NeuReduce can only simplify a handful of expressions in total. One limitation is that it can only work with inputs up to 100 characters; however, our artificial MBAs with a rewriting bound of 20 contain, on average, $7,960$ characters (after removing all

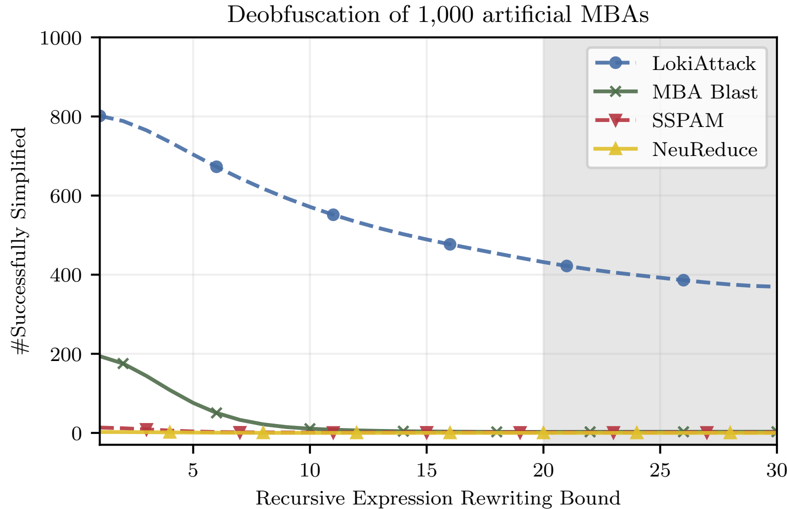


Figure 3: Number of artificial MBAs that have been successfully simplified (averaged over 5 different core semantics). The gray-shaded area marks the recursive rewriting bounds randomly picked by Loki for our regular MBAs.

whitespaces). We have tried a similar approach as we have employed for MBA-Blast, however, found it does not improve its accuracy. Studying their dataset used to train the model, we believe that their approach heavily overfits on the training data, a set of simple and short (on average 75 characters without whitespace) MBAs. SSPAM fails to deal with the highly recursive nature of our MBAs, frequently hitting the stack recursion limit. MBA-Blast performs better and manages to simplify a number of simple MBAs. However, the success rate of all tools decreases with a higher term rewriting bound. Loki's default is to use a random recursive rewriting bound between 20 and 30, for which all but LokiAttack fail to simplify basically any MBA. For example, MBA-Blast simplifies only 157 of $55,000$ MBAs for Loki's recursive rewriting bounds, $[20, 30]$. While the success rate of LokiAttack may seem high, recall that we artificially weakened these MBAs by excluding a number of operations and removing all constants; Experiment 8 evaluates how LokiAttack with the symbolic execution plugin performs on our regular MBAs.

Semantic Attacks. Semantic attacks such as program synthesis exploit the low semantic depth of individual core semantics. We evaluate the impact of our superoperators on these attacks. First, we analyze the average semantic complexity of core semantics with and without superoperators. Then, we perform a high-level experiment

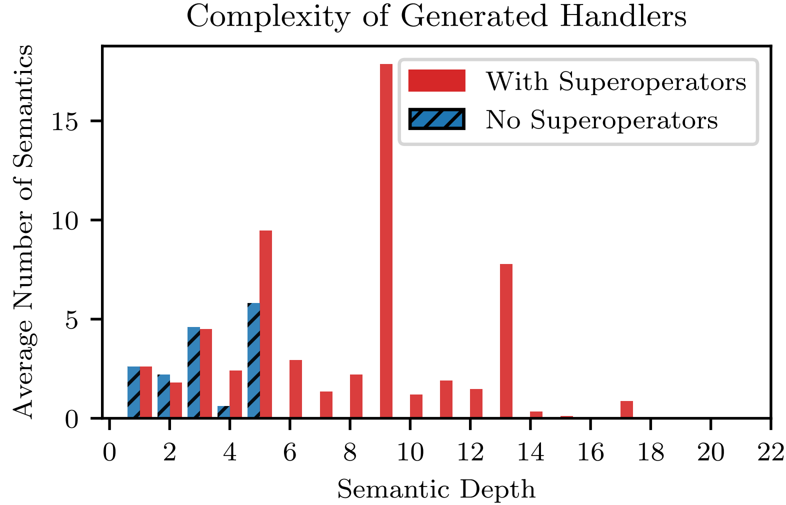


Figure 4: The distribution of core semantics with and without superoperators.

to measure the general limits of synthesis-based approaches. Finally, we demonstrate that our superoperators withstand synthesis-based attacks on the binary level. Note that we only consider a dynamic attacker in the following, as knowing a value for k is a prerequisite for any reasonable semantic attack. A static attacker would only learn random behavior, as the key encodings are only valid for a predefined set of keys.

Experiment 11: Complexity of Core Semantics. *To evaluate our superoperators' distribution and their impact on the complexity of core semantics, i. e., their semantic depth, we create* 1,000 *obfuscated binaries* without *superoperators as a baseline for each benchmarking target (cf. Section 6.1) and* 1,000 *binaries* with *superoperators. We compare the two sets on the average number of unique core semantics and their semantic depths. To simplify evaluation, no MBAs are used.*

Without superoperators, each binary on average contains 15.8 core semantics. With superoperators, this number increases to 58.8. Additionally, Figure 4 shows that superoperators have a significantly higher semantic depth, usually ranging from 5 to 13 with a clearly visible peak at depth 9. Compared to obfuscation without superoperators, where only a few core semantics with semantically low complexity are used, superoperators increase the number of unique core semantics and their semantic depth notably. This makes the task of synthesizing semantics more difficult.

Experiment 12: Limits of Program Synthesis. *We evaluate how the success rates of program synthesis relate to semantic complexity. We generate* 10,000 *random expres-*

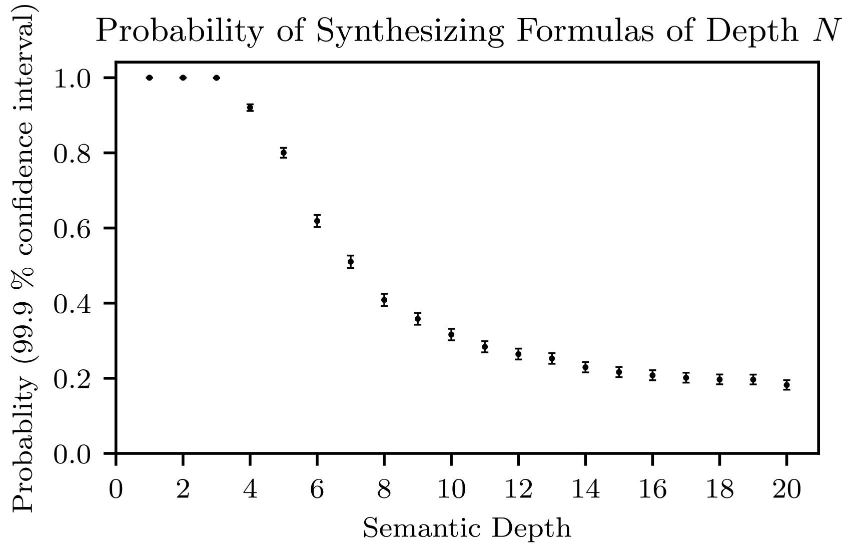


Figure 5: The probability to synthesize a valid candidate for formulas of depth N. The error bars are calculated as the 99.9% confidence interval for the true probability.

sions for each semantic depth between 1 *and* 20 *and measure how many of them can be synthesized successfully. Modeling our function* f*, we use* Syntia*'s grammar [31] to generate random expressions depending on three variables. Based on the authors' guidance, we set* Syntia*'s configuration vector to* (1.5, 50000, 20, 0) *and use it to synthesize each expression.*

Figure 5 shows that simple expressions can be synthesized quite easily; at a semantic depth of 7, only ~50% can be synthesized. For larger semantic depths, it becomes increasingly unlikely to synthesize expressions. Given our results from Experiment 11, we conclude that our superoperators produce core semantics of sufficient depth to impede program synthesis.

Experiment 13: Superoperators on Binary Level. *To evaluate the impact of program synthesis, we assume a dynamic attacker has extracted a handler's core semantics (for* Loki*: associated with a known value of* k*). They then use* Syntia*—configured as in Experiment 12—to learn an expression having the same input-output behavior. We create* 400 *obfuscated binaries (without MBAs, with superoperators) for each benchmarking target (cf. Section 6.1), randomly pick* 10,000 *core semantics and measure* Syntia*'s success rate.*

Overall, using Syntia as a stage 2 plugin on top of LokiAttack, we managed to synthesize 1,888 (~19%) of Loki's expressions and 6,779 (~68%) of Tigress' ex-

pressions. On average, it took 157s to synthesize an expression for Loki and 144s for Tigress. The results show that—while both designs employ superoperators—it is crucial how these superoperators are crafted. As outlined in Section 2.1, Tigress usually includes independent core semantics, allowing the attacker to split the superoperators into multiple smaller synthesis tasks, each of low semantic complexity. On the other hand, Loki's design ensures that its superoperators cannot be split into smaller tasks but have high semantic depths. In summary, Loki is the first obfuscation design showing sufficient protection against program synthesis, an attack vector all state-of-the-art obfuscators fail to account for. Given Loki's synergy effects and high resilience against syntactic simplification approaches, semantic deobfuscation techniques remain an attacker's last resort. However, even when using program synthesis, arguably the strongest semantic attack, an attacker can only recover less than a fifth of Loki's core semantics.

7 Discussion

Overhead. Table 2 indicates that the overhead of code obfuscation is generally excessive. However, this cost is accepted in practice because only small, critical parts of the whole program need to be protected (e. g., proprietary algorithms, API accesses, or licensing-related code). As a result, the overhead has to be seen in relation to the whole program. As our case study shows for libdvdcss, using obfuscation only for critical, well-chosen code parts has no negative impact on the usability of the respective program (here VLC).

MBA Database. Assuming an attacker intends to symbolically simplify MBAs, they may benefit from using a lookup table mapping complex MBAs to simpler expressions. This approach is effective for state-of-the-art obfuscators such as Tigress that only use a limited number of hardcoded rewriting rules (cf. Experiment 9). Loki, in contrast, is the first obfuscator that employs a generic approach to synthesize highly diverse MBAs, resulting in a large number of MBAs (stored in a database for performance reasons). Users of Loki can keep their MBA database (including the synthesis limit up to which MBAs were synthesized) private. In fact, users could choose arbitrary lower and upper limits as well as completely different grammars to create an MBA database. Without knowing the parameters, a re-creation of the database is not feasible. That

said, even in cases where an attacker is in possession of the MBA database, there is no straightforward process to reverse the recursively generated expression (cf. Section 4.4).

Attacker Model. Our evaluation assumes a strong attacker model with significant domain knowledge and access to all kinds of static and dynamic analyses. In practice, an attacker is often weaker. Especially without prior knowledge about the given obfuscation techniques, the usage of additional techniques (e. g., VM bytecode blinding [31] or range dividers [12]) and other countermeasures (e. g., self-modifying code [11] or anti-debugging techniques [81]) complicates analysis.

Human Attacker. Ultimately, code obfuscation schemes are usually broken by human analysts [21]. This is partly because humans excel at recognizing patterns and adapt to the given obfuscation [82]. Collberg et al. [1] define *potency* to denote how *confusing* an obfuscation is for a human analyst. Due to the difficulty of measuring a human's capability with regard to deobfuscation, we restrict our evaluation to automated attacks. We argue that without automated techniques, analysis becomes subjectively harder. Nevertheless, we believe that pattern matching might be the most potent attack on our approach. While we use a fixed structure, we argue that our MBAs remove identifiable patterns. Still, we are not aware of an adequate way of measuring this. However, even if we assume that a human attacker breaks one obfuscated instance, other instances remain hard. This is as our design samples from large search spaces for its critical components, providing significant diversity for MBAs and superoperators. In summary, we expect Loki to perform reasonably well against human attackers even if this cannot be easily quantified.

8 Related Work

Over the years, a large number of obfuscation techniques were proposed [1, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17]. Many of these techniques are orthogonal to our work and focus on one specific transformation. For an overview over the field of obfuscation, we refer the interested reader to the overview by Banescu and Pretschner [83]. In the following, we discuss techniques closest to our work.

MBA. Zhou et al. introduced the concept of Mixed Boolean-Arithmetic (MBA) to hide constants and calculations within complex expressions. While conceptually simple, this approach proved effective against many analysis techniques, such as symbolic execution. As a consequence, a number of approaches towards deobfuscating MBAs

were proposed, including pattern matching (SSPAM [36]), symbolic simplification using a Boolean expression solver (arybo [37]), program synthesis (Syntia [31], Xyntia [45], QSynth [32]), machine learning (NeuReduce [46]), and algebraic simplification (MBA-Blast [47]). While those techniques are effective against common MBAs, Loki's generic approach to synthesize diverse MBAs produces expressions resilient against such attacks (cf. Section 6).

VM Obfuscation. Our prototype implementation Loki uses a VM-based architecture to showcase our techniques. However, we make no attempt at obfuscating the VM structure itself, which we consider orthogonal to our work. Examples for such work include *virtual code folding*, where the mapping between opcodes and individual handlers is obfuscated to impede static analyses [18, 19, 20, 84]. While they use dynamic keys to determine the next handler, we use keys within our handlers to select a specific core semantics. With regard to deobfuscation, approaches such as VMHunt [85], VMAttack [86], and others [26, 34] may succeed in reconstructing Loki's VM structure (similar to LokiAttack). However, they cannot recover individual handler semantics, since they rely on techniques such as symbolic execution and backward slicing, for which Loki is resilient against by design.

Thwarting Symbolic Execution. With regard to thwarting symbolic execution-based deobfuscation approaches, early work by Sharif et al. [87] already proposed key-based encodings to make path exploration infeasible. Later approaches extend on this work by introducing multi-level opaque predicates (so-called *range dividers*) [12] or artificial loops [16]. Loki extends these ideas: it does not only make path exploration infeasible, but also prevents symbolic simplification attacks due to its MBAs.

Thwarting Program Synthesis. Program synthesis is one of the most powerful attack vectors [31, 45]. Concurrent work [45] proposes a search-based program synthesis approach outperforming Syntia. However, the authors note that merging handlers and increasing a handler's semantic complexity proved effective in thwarting such attacks. This is in line with our evaluation.

9 Conclusion

In this paper, we present and extensively evaluate a set of novel and generic obfuscation techniques that, in combination, succeed to thwart automated deobfuscation attacks.

Acknowledgments

We would like to thank our shepherd Kevin Butler and the anonymous reviewers for their valuable comments and suggestions. We also thank Marcel Bathke, Nils Bars, Berthold Dors, and Robert Stark for their help.

References

[1] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[2] Weiyun Lu, Bahman Sistany, Amy Felty, and Philip Scott. Towards Formal Verification of Program Obfuscation. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020.

[3] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[4] Fukutomo Nakanishi, Giulio De Pasquale, Daniele Ferla, and Lorenzo Cavallaro. Intertwining ROP Gadgets and Opaque Predicates for Robust Obfuscation. *CoRR*, abs/2012.09163, 2020.

[5] Abdelrahman Eid. Reverse Engineering Snapchat (Part I): Obfuscation Techniques. `https://hot3eed.github.io/snap_part1_obfuscations.html`.

[6] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *International Workshop on Information Security Applications*, 2007.

[7] Gregory Wroblewski. General Method of Program Code Obfuscation. In *International Conference on Software Engineering Research and Practice (SERP)*, 2002.

[8] Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: An Obfuscation Approach using Probabilistic Control Flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[9] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage Binary Code Obfuscation using Improved Virtual Machine. In *International Conference*

on Information Security, pages 168–181. Springer, 2011.

[10] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control Flow based Obfuscation. In *ACM Workshop on Digital Rights Management*. ACM, 2005.

[11] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software Protection through Dynamic Code Mutation. In *International Workshop on Information Security Applications*. Springer, 2005.

[12] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code Obfuscation against Symbolic Execution Attacks. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[13] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear Obfuscation to Combat Symbolic Execution. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[14] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – Software Protection for the Masses. In *ACM Workshop on Software PROtection (SPRO)*, 2015.

[15] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. Manufacturing Resilient Bi-Opaque Predicates against Symbolic Execution. In *Conference on Dependable Systems and Networks (DSN)*, 2018.

[16] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[17] Pietro Borrello, Emilio Coppa, and Daniele Cono D'Elia. Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation. In *Conference on Dependable Systems and Networks (DSN)*, 2021.

[18] Jae Hyuk Suk and Dong Hoon Lee. VCF: Virtual Code Folding to Enhance Virtualization Obfuscation. *IEEE Access*, 8, 2020.

[19] Chao Xue, Zhanyong Tang, Guixin Ye, Guanghui Li, Xiaoqing Gong, Wei Wang, Dingyi Fang, and Zheng Wang. Exploiting Code Diversity to Enhance Code Vir-

tualization Protection. In *International Conference on Parallel and Distributed Systems*, 2018.

[20] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. DynOpVm: VM-based Software Obfuscation with Dynamic Opcode Mapping. In *International Conference on Applied Cryptography and Network Security*, 2019.

[21] Rolf Rolles. Unpacking Virtualization Obfuscators. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[22] Oreans Technologies. Themida – Advanced Windows Software Protection System. `https://www.oreans.com/Themida.php`.

[23] VMProtect Software. VMProtect Software. `https://vmpsoft.com/`.

[24] Sony DADC. SecuROM Software Protection. `https://www2.securom.com/Digital-Rights-Management.68.0.html`.

[25] Denuvo Software Solutions GmbH. Denuvo Anti-Tamper. `http://www.denuvo.com`.

[26] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy*, 2009.

[27] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, 2015.

[28] Peter Garba and Matteo Favaro. SATURN – Software Deobfuscation Framework Based On LLVM. In *ACM Workshop on Software PROtection (SPRO)*, 2019.

[29] Babak Yadegari and Saumya Debray. Bit-level Taint Analysis. In *IEEE Conference on Source Code Analysis and Manipulation*, 2014.

[30] Babak Yadegari and Saumya Debray. Symbolic Execution of Obfuscated Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[31] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *USENIX Security Symposium*, 2017.

[32] Robin David, Luigi Coniglioi, and Mariano Ceccato. QSynth – A Program Synthesis based Approach for Binary Code Deobfuscation. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2020.

[33] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-Based Approach. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[34] Johannes Kinder. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *IEEE Working Conference on Reverse Engineering (WCRE)*, 2012.

[35] Lucas Barhelemy, Ninon Eyrolles, Guenaël Renault, and Raphaël Roblin. Binary Permutation Polynomial Inversion and Application to Obfuscation Techniques. In *ACM Workshop on Software PROtection (SPRO)*, 2016.

[36] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In *ACM Workshop on Software PROtection (SPRO)*, 2016.

[37] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *GreHack Conference*, 2016.

[38] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *IEEE Symposium on Security and Privacy*, 2017.

[39] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2017.

[40] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization. In *International Conference on Information and Communications Security*, 2017.

[41] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2018.

[42] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-Taint-Analysis: Practical Evasion Techniques against Information Flow-based Malware Defense. Technical report, Secure Systems Lab, Stony Brook University, 2007.

[43] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices. In *International Conference on Security and Cryptography (SECRYPT)*, 2013.

[44] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In *USENIX Security Symposium*, 2017.

[45] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[46] Weijie Feng, Binbin Liu, Dongpeng Xu, Qilong Zheng, and Yun Xu. NeuReduce: Reducing Mixed Boolean-Arithmetic Expressions by Recurrent Neural Network. In *Conference on Empirical Methods in Natural Language Processing: Findings*, 2020.

[47] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In *USENIX Security Symposium*, 2021.

[48] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Technical Report: Hardening Code Obfuscation Against Automated Attacks. `https://arxiv.org/abs/2106.08913`, 2022.

[49] Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1995.

[50] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. Boosting SMT Solver Performance on Mixed-bitwise-arithmetic Expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[51] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution

(But Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*, 2010.

[52] Mark Weiser. Program Slicing. In *International Conference on Software Engineering (ICSE)*, 1981.

[53] Sebastian Danicic and Michael R. Laurence. Static Backward Slicing of Non-Deterministic Programs and Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(3):11:1–11:46, 2018.

[54] Julien Vanegue, Sean Heelan, and Rolf Rolles. SMT Solvers in Software Security. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[55] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Conference on Dependable Systems and Networks (DSN)*, pages 177–186. IEEE, 2008.

[56] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. Advanced or Not? A Comparative Study of the Use of Anti-Debugging and Anti-VM Techniques in Generic and Targeted Malware. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 323–336, 2016.

[57] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2016.

[58] Sumit Gulwani. Dimensions in Program Synthesis. In *International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2010.

[59] Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *ACM Sigplan Notices*, 2006.

[60] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and Extensible Equality Saturation. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[61] Christian Collberg. The Tigress C Diversifier/Obfuscator. `http://tigress.cs.arizona.edu/`.

[62] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1989.

[63] Michał Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[64] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[65] Aki Helin. Radamsa: A General-purpose Fuzzer. https://github.com/aoh/radamsa.

[66] Michael Eddington. Peach Fuzzer: Discover Unknown Vulnerabilities. https://www.peach.tech/.

[67] The LLVM Project. The LLVM Compiler Infrastructure. https://llvm.org/.

[68] Paul Klint. Interpretation Techniques. *Software, Practice and Experience*, 11(9):963–973, 1981.

[69] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1998.

[70] Microsoft Research. The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[71] Intel Corporation. Pin – A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[72] CEA IT Security. Miasm – Reverse Engineering Framework. https://github.com/cea-sec/miasm.

[73] Florent Saudel and Jonathan Salwan. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2015.

[74] Robin Verton. RC4. https://gist.github.com/rverton/a44fc8ca67ab9ec32089, 2015.

[75] Jasin Bushnaief. SHA-1. https://gist.github.com/rverton/a44fc8ca67ab9ec32089, 2016.

[76] Christian Collberg. Performance vs. Security. https://tigress.wtf/blog.html.

[77] VideoLAN. libdvdcss. `https://www.videolan.org/developers/libdvdcss.html`.

[78] VideoLAN. VLC media player. `https://www.videolan.org/`.

[79] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *International Conference on Computer Aided Verification*, 2000.

[80] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[81] Michael N Gagnon, Stephen Taylor, and Anup K Ghosh. Software Protection through Anti-Debugging. *IEEE Security & Privacy*, 5(3):82–84, 2007.

[82] B. Dang, A. Gazet, E. Bachaalany, and S. Josse. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Wiley, 2014.

[83] Sebastian Banescu and Alexander Pretschner. A Tutorial on Software Obfuscation. *Advances in Computers*, 108:283–353, 2018.

[84] Jae-Yung Lee, Jae Hyuk Suk, and Dong Hoon Lee. VODKA: Virtualization Obfuscation Using Dynamic Key Approach. In *International Workshop on Information Security Applications*, 2018.

[85] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partially-virtualized Binary Code Simplification. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[86] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. VMAttack: Deobfuscating Virtualization-based Packed Binaries. In *Availability, Reliability and Security (ARES)*, 2017.

[87] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.