Linköping Studies in Science and Technology
Dissertations, No. 1993

Scalable Dynamic Analysis of Binary Code

Ulf Kargén

Linköping Studies in Science and Technology
Dissertations, No. 1993

Scalable Dynamic Analysis of Binary Code

Ulf Kargén

Linköping University
Department of Computer and Information Science
Division of Database and Information Techniques
SE-581 83 Linköping, Sweden

Linköping 2019

Edition 1:1

© Ulf Kargén, 2019

ISBN 978-91-7685-049-7

ISSN 0345-7524

URL <http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-157626>

Published articles have been reprinted with permission from the respective copyright holder.

Typeset using X_YTEX

Printed by LiU-Tryck, Linköping 2019

POPULÄRVETENSKAPLIG SAMMANFATTNING

Programvara skrivs oftast i så kallade *högnivåspråk*, som möjliggör en tydlig och strukturerad representation av hur programmet fungerar. Detta gör det dels enkelt för programmerare att läsa programkoden och resonera om hur den fungerar, och dels att skapa avancerade verktyg för automatisk analys av programkod, t.ex. för att upptäcka fel och säkerhetsbrister. Analysverktyg kan använda sig av antingen *statisk analys*, som undersöker programmets kod utan att köra det, eller *dynamisk analys*, som observerar hur programmet beter sig under körning. En begränsning hos det senare tillvägagångssättet är att endast kod som faktiskt körs kan analyseras.

För att ett program skall kunna köras på en dator måste det först översättas från högnivåspråket det är skrivet i till *binärkod*, som består av instruktioner som är anpassade för att tolkas och exekveras av datorns processor. Binärkod utgör en minimal representation av ett program och saknar det mesta av informationen från högnivåspråket, vilket gör binärkod mycket svår att läsa och analysera. Att kunna analysera program direkt på binärkods-nivån har dock många viktiga tillämpningar, som exempelvis analys av skadlig kod och säkerhetsgranskning av programvara som endast distribueras i binärkodsform. Även om framsteg gjorts när det gäller att med approximativa metoder återskapa viss högnivåinformation från binärkod har statisk analys på binärkods-nivån mycket begränsad noggrannhet. För precis analys av binärkod krävs därför dynamisk analys, som inte på samma sätt begränsas av bristen på högnivåinformation. Detaljerad dynamisk analys kräver dock att stora mängder information om ett programs beteende under körning spelas in, för att senare kunna analyseras. Detta skapar i stället utmaningar när det gäller *skalbarhet*, dvs. att kunna spela in och bearbeta all information utan att det kräver orimligt mycket tid eller minnesutrymme.

I den här avhandlingen presenteras skalbara versioner av två kraftfulla metoder för dynamisk analys vars praktiska användbarhet tidigare begränsats av bristande skalbarhet. Den första metoden är *dynamisk slicing*, som används för att i detalj studera beroenden mellan olika delar av ett program, och som kan användas dels som ett kraftfullt hjälpmedel vid felsökning och dels som bas för andra programanalysmetoder. Mängden information som måste spelas in för att utföra dynamisk slicing är dock ofta så stor att den inte får plats i en dators primärminne. Existerande metoder använder därför olika tekniker för att begränsa mängden information som explicit behöver sparas i minnet, vilket tyvärr i stället gör dem mycket långsamma. Vi undviker denna prestandaförlust genom att optimera våra algoritmer för att möjliggöra snabb lagring och åtkomst av informationen från datorns hårddisk, vilket gör att mängden primärminne inte längre är en begränsning. För att undvika att hårddiskens stora latens-tid (tiden det tar från det att information efterfrågas tills dess att den börjar läsas från disken) blir ett problem har vi optimerat vår metod genom att utnyttja s.k. tidsmässig lokalitet (*eng. temporal locality*) i program.

Vårt andra bidrag inom skalbar dynamisk analys är en snabb metod för s.k. ensning av instruktionsspår (*eng. instruction trace alignment*). Ett instruktionsspår utgörs av en komplett kronologiskt ordnad sekvens av alla instruktioner som exekverats under en programkörning. Ensning av två instruktionsspår innebär att, för varje instruktion i det ena spåret, hitta den motsvarande instruktionen i det andra spåret. Genom att göra en sådan sida-vid-sida-jämförelse kan likheter och skillnader i beteende hos två liknande program kvantifieras och studeras i detalj, vilket är användbart vid exempelvis analys av skadlig kod. Eftersom traditionella metoder för ensning av sekvenser inte är tillräckligt skalbara för att kunna användas på instruktionsspår har vi utvecklat en egen approximativ metod, som så vitt vi vet är den första generella metoden för ensning av realistiskt långa instruktionsspår.

Vi har också haft som målsättning att praktiskt tillämpa de metoder vi utvecklat för skalbar dynamisk analys. Specifikt har vi tillämpat vårt tillvägagångssätt för skalbar dynamisk slicing i två nya metoder för *fuzzing*. Fuzzing är en programtestningsteknik som används flitigt inom industrin för att upptäcka bl.a. säkerhetshål och bristande robusthet hos programvara. Metoden fungerar genom att skicka semi-slumpmässigt genererad data till programmet för att testa att det kan hantera felaktigt formaterad indata. Vårt första bidrag inom fuzzing utnyttjar dynamisk slicing och mutation av binärkod för att automatiskt transformera ett existerande program, som kan generera indata till programmet som skall testas, till en testgenerator för fuzzing. Exempelvis kan ett program för att generera PDF-filer automatiskt transformeras till en testgenerator för PDF-läsare. Vårt andra arbete inom fuzzing använder dynamisk slicing för att förbättra fuzzers som använder sig av indatamutation. Sådana fuzzingverktyg genererar testdata genom att göra slumpmässiga förändringar i en korrekt indatafil. Med hjälp av dynamisk slicing kan vi upptäcka vilka delar av en muterad indatafil som påverkar en hittills utforskad del av programmet som skall testas. Genom att rikta fler mutationer till den delen ökar sannolikheten att snabbare hitta fel i programmet, jämfört med att helt slumpmässigt välja ut vilka delar av indatafilen som skall muteras.

ABSTRACT

In recent years, binary code analysis, i.e., applying program analysis directly at the machine code level, has become an increasingly important topic of study. This is driven to a large extent by the information security community, where security auditing of closed-source software and analysis of malware are important applications. Since most of the high-level semantics of the original source code are lost upon compilation to executable code, static analysis is intractable for, e.g., fine-grained information flow analysis of binary code. Dynamic analysis, however, does not suffer in the same way from reduced accuracy in the absence of high-level semantics, and is therefore also more readily applicable to binary code. Since fine-grained dynamic analysis often requires recording detailed information about every instruction execution, scalability can become a significant challenge. In this thesis, we address the scalability challenges of two powerful dynamic analysis methods whose widespread use has, so far, been impeded by their lack of scalability: *dynamic slicing* and *instruction trace alignment*. Dynamic slicing provides fine-grained information about dependencies between individual instructions, and can be used both as a powerful debugging aid and as a foundation for other dynamic analysis techniques. Instruction trace alignment provides a means for comparing executions of two similar programs and has important applications in, e.g., malware analysis, security auditing, and plagiarism detection. We also apply our work on scalable dynamic analysis in two novel approaches to improve *fuzzing* — a popular random testing technique that is widely used in industry to discover security vulnerabilities.

To use dynamic slicing, detailed information about a program execution must first be recorded. Since the amount of information is often too large to fit in main memory, existing dynamic slicing methods apply various time-versus-space trade-offs to reduce memory requirements. However, these trade-offs result in very high time overheads, limiting the usefulness of dynamic slicing in practice. In this thesis, we show that the speed of dynamic slicing can be greatly improved by carefully designing data structures and algorithms to exploit *temporal locality* of programs. This allows avoidance of the expensive trade-offs used in earlier methods by accessing recorded runtime information directly from secondary storage without significant random-access overhead. In addition to being a standalone contribution, scalable dynamic slicing also forms integral parts of our contributions to fuzzing. Our first contribution uses dynamic slicing and binary code mutation to automatically turn an existing executable into a test generator. In our experiments, this new approach to fuzzing achieved about an order of magnitude better code coverage than traditional mutational fuzzing and found several bugs in popular Linux software. The second work on fuzzing presented in this thesis uses dynamic slicing to accelerate the state-of-the-art fuzzer AFL by focusing the fuzzing effort on previously unexplored parts of the input space.

For the second dynamic analysis technique whose scalability we sought to improve — instruction trace alignment — we employed techniques used in speech recognition and information retrieval to design what is, to the best of our knowledge, the first general approach to aligning realistically long program traces. We show in our experiments that this method is capable of producing meaningful alignments even in the presence of significant syntactic differences stemming from, for example, the use of different compilers or optimization levels.

The work was supported by the Swedish Graduate School in Computer Science (CUGS) and the Excellence Center at Linköping – Lund in Information Technology (ELLIIT)

Acknowledgments

First and foremost, my deepest gratitude goes to my supervisor Professor Nahid Shahmehri for giving me the opportunity to do a PhD at ADIT. Thank you for your guidance and support, and for the freedom you have allowed me to explore the wilderness that is computer science (and for occasionally sending out a search party when I strayed too far off the beaten path).

I would also like to thank my secondary supervisor Professor Patrick Lambrix.

I thank my colleagues at ADIT and IDA for contributing to a friendly and stimulating workplace. A special mention goes to Anna Vapen for introducing me to life as a PhD student, for all weird lunch-time discussions, and for being a good friend in general. I would also like to thank Adi, Arian, Felipe, Klervie, Mikaela, Rodrigo and the rest of the SaS lunch crew (past and present) for brightening my days with interesting discussions.

I am grateful to Brittany Shahmehri for her excellent work proofreading the thesis.

Finally, I would like to thank my parents, my brother, and the sheep for providing often much needed (and sometimes less so) distractions from work.

*Ulf Kargén
July 2019
Linköping, Sweden*

Contents

Abstract	iii
Acknowledgments	vii
Contents	viii
1 Introduction	1
1.1 The Need for Binary Code Analysis	2
1.2 Binary Code Analysis Challenges	3
1.3 Security Testing using Fuzzing	3
1.4 Problem Formulation	5
1.5 Research Method	6
1.6 Contributions	7
1.7 Thesis Outline	8
1.8 List of Publications	9
2 Background	11
2.1 Static Analysis of Binary Code	11
2.2 Dynamic Analysis of Binary Code	14
3 Related Work	21
3.1 Dynamic Information Flow Analysis	21
3.2 Fuzzing	27
3.3 Trace Alignment and Binary Code Matching	35
4 Summary of Contributions	41
4.1 Scalable Dynamic Slicing	41
4.2 Scalable Instruction Trace Alignment	45
4.3 Improving Fuzzing using Dynamic Slicing	47
5 Conclusions and Future Work	53
5.1 Conclusions	54
5.2 Future Work	55

Bibliography	59
Publications	73
Paper I Efficient Utilization of Secondary Storage for Scalable Dynamic Slicing	75
Paper II Towards Robust Instruction-Level Trace Alignment of Binary Code	99
Paper III Turning Programs against Each Other: High Coverage Fuzz-Testing using Binary-Code Mutation and Dynamic Slicing	127
Paper IV Speeding Up Bug Finding using Focused Fuzzing	155

Chapter 1

Introduction

Software is everywhere. Aside from the enormous impact the Internet and, for example, social media have had on our everyday lives, the accelerating digitalization of all aspects of society also means that software is currently becoming a crucial part of every company's business model. In order to stay competitive and to keep up with rapidly evolving customer demands, many businesses are quickly moving from fixed-function hardware solutions to complex software solutions, which enable more flexibility and faster development of new features. Advanced automation technology, like self-driving cars, as well as the advent of the Internet of Things era, means that complex software solutions will soon proliferate into every aspect of our lives. Inevitably, however, with the increasing complexity of software systems comes an increased risk of dangerous software flaws. For example, the WannaCry ransomware attack in 2017 was made possible by a security bug in Microsoft Windows that allowed the malware to spread quickly from computer to computer. The attack crippled hundreds of thousands of computers and caused losses upwards of \$4 billion according to some estimates [12]. Later the same year, the NotPetya malware used the same vulnerability to spread, causing damages estimated at up to \$10 billion [49]. With our increasingly connected and computerized society, it's not only digital assets that are at risk. Recent incidents such as the 2015 hacker attack on the Ukrainian power grid [115], or the intrusion into the control systems of a US water treatment plant in 2016 [29] highlight the risk of buggy software in critical infrastructure. Several car manufacturers have also recently been forced to initiate large recalls due to software bugs in safety-critical systems [51], or even remotely-exploitable bugs giving potential attackers control over a car's drivetrain [48].

With the increasing dependence on complex software in safety-critical systems, and ever more capable attackers in the form of specialized crime rings and nation-state run hacking groups, there is also an increasing need for more advanced program analysis methods. Such methods can aid both in discovering critical flaws in software before they can be exploited, and in dissecting

and understanding malicious software. This thesis makes contributions within the field of *dynamic analysis*, i.e. analyzing the behavior of software while it runs, and demonstrates applications of such methods for automated security and robustness testing using *fuzzing*. Our focus in the work presented here has been on analysis of binary code. In the following section, we elaborate on the need for robust analysis methods at the binary code level.

1.1 The Need for Binary Code Analysis

Recent years have seen significant developments in program analysis methods that work directly on the machine code level. To a large extent, the interest in binary-code analysis has been driven by the security community, where the ability to analyze software in the absence of source code is important for two primary reasons. Firstly, many software products are distributed only in binary form, either as closed-source software packages or as built-in firmware. Binary code analysis allows independent security evaluation of such software, at least to some degree. For many consumer products, independent evaluation of quality and safety properties has been an established practice for many years: Cars can be subjected to crash tests, or to manual inspection of mechanical parts, to evaluate their safety; home appliances can be dismantled and evaluated for compliance with electrical safety standards; clothes can be tested for traces of hazardous chemicals, and so on. Without powerful binary-analysis tools, closed-source software remains a "black box", leaving consumers with little choice other than to blindly trust the vendor's claims about, for example, its security. While binary analysis has previously been met with some suspicion from software vendors, on the grounds of fear of intellectual-property theft, many vendors now appear to recognize binary analysis as an acceptable middle ground between complete absence of transparency and going full open-source. A recent example of this is Microsoft, which is now paying monetary awards ("bug bounties") for security flaws found in their closed-source products via binary inspection [81].

The second driving force behind the interest in binary analysis within the security community is malware. Source code is rarely ever available for in-the-wild malware, and malware authors also routinely apply obfuscation techniques to harden malicious binaries against analysis, in order to delay detection and development of countermeasures.

Aside from applications in security, there are also several other reasons to perform analysis on the machine-code level. One reason is *ease of deployment*: If an analysis tool is to be used on many different software projects, perhaps developed using different languages, using different complex build environments, or using third-party libraries without source code, it may be more convenient to perform analysis directly on the compiled binaries. For example, the automated security testing system SAGE [46] was developed for

in-house use at Microsoft, but for the above-mentioned reasons the authors still opted for performing all analysis on the binary code level. Another reason for performing analysis on the final binaries rather than on the source code is the "*what you see is not what you execute*" phenomenon. Compiler bugs, unforeseen side-effects of optimization, or post-compilation transformations may alter the semantics of the executable code from those of the source code [6]. Finally, source code may not be available for *legacy software*, making binary code analysis the only option.

1.2 Binary Code Analysis Challenges

While there are many applications of binary code analysis, there are also significant challenges associated with it, compared to source code analysis. Since much of the semantic information is lost when source code is compiled to machine code, static analysis on binaries is generally difficult. For example, machine code lacks structured control flow, data types, and variable names. High-level data structures, like classes and records, are also lost during compilation. While most of the above can be partially recovered from binaries using heuristic methods [26, 108, 76, 73, 41], the results of such methods are typically incomplete. Therefore, the accuracy of static analyses built on top of recovered high-level program characteristics is often limited.

Dynamic analysis is, in some ways, a more tractable approach for analyzing binaries, since its accuracy is not limited in the same way by lack of high-level semantics in machine code. If every executed instruction can be tracked dynamically, it is still possible to record an exact representation of the data or control flow exhibited during an execution of a program. Aside from the obvious limitation that only the behavior of code that actually executes can be studied, a severely limiting factor for such fine-grained dynamic analysis is scalability. A modern CPU can execute several billion instructions per second, meaning that even for fairly short program executions, dynamic analysis can generate enormous amounts of data.

1.3 Security Testing using Fuzzing

Fuzzing is a conceptually simple random testing technique: Semi-valid inputs are generated by a piece of software called a *fuzzer*. These inputs are then fed to the program under test (PUT), while the behavior of the PUT is monitored to detect failures. Fuzzing has proven remarkably effective at uncovering low-level implementation errors like buffer overflows and other memory-corruption bugs. Such errors can often be exploited by attackers to take control of a running process and break into a computer system. Since inputs are (semi)randomly generated, fuzzers typically do not require source code for the PUT. The ability to find security vulnerabilities in closed-source

software contributed significantly to making fuzzing popular among security practitioners during the early 2000s. Today, however, fuzzing is also heavily employed in-house among many large software vendors [5, 85, 57].

The key challenge when designing a fuzzer is to find a good balance between generating inputs that, on the one hand, adhere sufficiently to the input format expected by the PUT to not be rejected by early sanity checks, but that are, on the other hand, sufficiently "wrong" to be able to trigger faults. Within the design space of fuzzers, some methods put a greater emphasis on fast test execution rates using random exploration, while others employ program analysis and reasoning to produce inputs that are less likely to be rejected early by the PUT, at the expense of a slower testing rate. Most existing fuzzers can be classified according to the following taxonomy:

- **Blackbox fuzzers** "blindly" generate inputs, and do not rely on any knowledge of the internal workings of the PUT, nor any feedback other than observable program failures (i.e. crashes). The two most common types of blackbox fuzzers are *mutational* and *generational* fuzzers. The former works by applying small random mutations to a valid input, called the *seed input*. The rationale for the seed input is that a slightly mutated valid input is more likely to (approximately) adhere to the correct input format than a completely random input. However, blackbox mutational fuzzers often achieve very limited code coverage in practice. Generational fuzzers instead rely on a pre-defined grammar to generate semi-valid inputs. While the generational approach typically achieves superior coverage compared to mutational fuzzers, the large up-front manual effort of crafting the input grammar may make generational fuzzing infeasible for PUTs with highly complex input formats.
- **Whitebox fuzzers** employ heavyweight program analysis to observe the execution path of the PUT and record branch constraints. An SMT (satisfiability modulo theory) solver is then used to generate an input that is guaranteed to lead execution down a previously unexplored path. While whitebox fuzzers can achieve much better code coverage compared to random input-space exploration, their scalability is limited. Since solving SMT queries is an NP-complete problem, constraints for "deep" program paths may take an unacceptably long time to solve.
- Similar to mutational fuzzers, **greybox fuzzers** rely on random mutations of inputs. However, the random input-space exploration is guided by coverage feedback, such that generated inputs that improve code coverage are fed back to the fuzzer for further mutation. This way, the fuzzer implements an evolutionary algorithm of sorts, allowing it to gradually uncover new regions of the input space.

Whitebox fuzzers favor generating "high-quality" inputs at a slow rate, while blackbox fuzzers, due to their simplicity, can generate and execute test cases at a rapid rate. Greybox fuzzers are faster than whitebox fuzzers, since they use random exploration, but are slower than pure blackbox mutational fuzzers due to the coverage instrumentation applied to the PUT. In the following discussion, we will refer to fuzzing techniques that use random input generation simply as *random fuzzing*.

1.4 Problem Formulation

Fine-grained dynamic analysis can provide a wealth of information about the behavior of a binary program. However, this potential remains partially untapped due to scalability challenges. We have focused on two types of dynamic analysis which, historically, have not been widely adopted due to scalability issues: *Dynamic slicing* and *instruction trace alignment*.

Program slicing (or simply *slicing*) is a well-established technique that can aid in various program-comprehension tasks, and can also be used as a basis for other analyses. For a given program instruction I , referred to as the *slicing criterion*, slicing can be used to answer the questions "What is the set of instructions influenced by I ?" (forward slicing) and "What is the set of instructions that influence I ?" (backward slicing). These sets are referred to as *slices*. Slices can be computed both statically and dynamically. In static program slicing, the control flow and the data flow of the program code are analyzed to determine all code that *may* influence (or be influenced by) the slicing criterion [123, 10]. By contrast, dynamic slicing operates on a concrete execution of the program and computes slices based on dependencies that are actually exhibited between executed instructions [69, 2]. A dynamic slice is therefore always a subset¹ of the corresponding static slice, since static slicing needs to consider all possible execution paths involving the slicing criterion. While static slicing works well on source code, it is very hard to apply to binaries due to the aforementioned challenges with static analysis of machine code [66]. Dynamic slicing, however, is readily applicable to binary code. Since dynamic slicing requires tracking and recording of all dynamic dependencies during an execution, algorithms and data structures must be designed with scalability in mind. Therefore, our first research question has been:

RQ1: *How can scalable dynamic slicing of binary programs be realized?*

Given two program traces, i.e. recorded sequences of all executed instructions from two program runs, instruction trace alignment is the process of finding an (approximate) correspondence between individual instructions in each trace. Instruction trace alignment allows detailed comparison of the behavior

¹though not necessarily a strict subset

of, for example, two versions of the same program, and has applications in, e.g., malware analysis [62, 65], assessment of security vulnerabilities [60], and plagiarism detection [88]. However, existing general-purpose sequence alignment algorithms have quadratic time complexity, making them unsuitable for aligning real-life program traces, which are often millions of instructions long. While previous works have targeted special cases of the problem, like aligning traces of the same program with different inputs [60], or traces of syntactically modified (but semantically identical) versions of programs [138, 88], we instead focus on a more general formulation of the problem. Specifically, our aim has been a method that can align instruction traces both in the presence of syntactic and (minor) semantic differences between traced programs, as formulated in our second research question:

RQ2: *How can scalable trace alignment be achieved in the presence of syntactic or semantic differences between traced programs?*

Apart from developing new methods that address scalability challenges in the two aforementioned areas, our third aim has been to implement practical applications of our dynamic-analysis methods to improve security testing. Specifically, we have striven to ameliorate two major limitations of random fuzzing. Firstly, random fuzzing suffers from low code coverage due to lack of semantic insight. While whitebox fuzzing can achieve better coverage in theory, scalability remains a severely limiting factor. Generational fuzzing can achieve better code coverage due to manually-provided semantic insight, but at the expense of significant human effort. Therefore, our aim has been to retain the scalability of random fuzzing, while at the same time avoiding the effort of crafting input-grammars for generational fuzzing:

RQ3a: *How can the semantic insight of random fuzzing be improved without resorting to laborious manually-crafted input specifications?*

A second limitation of random fuzzing is that finding bugs or increasing code coverage may take a very long time for programs with longer execution times, or when seed inputs are large. For such cases, it is desirable to minimize the number of mutation/execution cycles by avoiding unnecessary mutation of redundant parts of inputs:

RQ3b: *How can the exploration of redundant or unimportant parts of the input be avoided in random fuzzing?*

1.5 Research Method

When addressing the above-mentioned research questions, we have used the following general approach: We begin with a literature survey to find existing approaches to solving the research problem or closely-related problems. Our proposed solution is then designed based on this survey and our own experience, taking into account the strengths and limitations of existing methods.

We then proceed to implement a prototype solution based on our design. A preliminary evaluation of the prototype implementation is performed, using a smaller set of test cases or smaller problem sizes. Based on the preliminary evaluation, the design and implementation may be iteratively improved. Lastly, the final implementation of the method is subjected to a more rigorous experimental evaluation.

When designing methods and algorithms, we have followed a problem-solving philosophy similar to *Occam's razor*, in that we have striven to make as few assumptions as possible about the properties of the programs being analyzed. A highly-specialized method may achieve better results than a general one for a subset of problems by incorporating many heuristics based on specific assumptions about the problem to be solved. However, this inevitably introduces a larger number of failure modes, i.e., situations where the method breaks due to unforeseen corner cases. Avoiding such failure modes is crucial when designing methods for use in adversarial settings. For example, if a program analysis method is to be used to analyze malicious code, it is important that it also generalizes to work for deliberately obfuscated binaries. Similarly, fuzzing often causes unexpected behavior in target programs. If an analysis method cannot handle code that behaves in a "non-typical" way, it cannot be used as part of a fuzzing pipeline.

To improve generality, we also strive to avoid tunable parameters when designing methods and algorithms. While unavoidable in some cases, parameters whose values must be determined empirically pose a threat to the external validity of a solution, since there is always a risk of "overfitting" to the specific test cases used for evaluation.

1.6 Contributions

The work presented within this thesis makes the following contributions:

- In addressing **RQ1** we explore a design space for dynamic slicing that, to the best of our knowledge, has not been studied before. The main challenge with applying dynamic slicing in practice is managing the huge data volumes produced by recording dynamic dependency information. Previous methods [136, 137] have focused on compressing this information so that it fits in RAM, thereby avoiding expensive random access to secondary storage. However, since the dependence information needs to be "unpacked" prior to use, such methods have high computational overhead. We instead show that secondary storage can in fact be used to store dependency information without significant I/O overhead, by exploiting temporal locality in programs. We present a method and associated data structures for achieving this in Paper I. We have also implemented another dynamic-slicing system based on similar design

principles, which is utilized in Paper IV and described briefly in Section 4.1.

- In Paper II we describe the first general method for approximate trace-alignment of binary programs, (**RQ2**). We show how to adopt the *Fast Dynamic Time Warping* algorithm for the case of instruction-trace alignment. Our method is general enough that it could potentially also be applicable to other related approximate sequence alignment problems.
- In Paper III, we show how the low human effort of mutational fuzzing can be married with the better preservation of correct input semantics (and thus better code coverage) of generational fuzzing (**RQ3a**). In essence, our method uses dynamic slicing and binary code mutation to turn an existing *generating program* into a generational fuzzer, without human intervention. Here, the term “generating program” refers to an existing program that can produce valid inputs to the PUT.
- Finally, we propose in Paper IV a way to achieve a better trade-off between *speed* and *thoroughness* in the state-of-the-art fuzzer AFL. By utilizing dynamic slicing to extend the notion of code coverage to individual input bytes, we can focus the fuzzing effort to more “promising” parts the of inputs (**RQ3b**).

While the first two contributions are directly related to scalable dynamic analysis of binary code, performing analysis on the binary code level and addressing scalability were also important goals during the work on our two contributions within fuzzing. Applying dynamic slicing directly at the binary-code level has two main advantages in our fuzzing applications. Firstly, an obvious benefit of not requiring source code for the analysis is that it also allows fuzzing of closed-source programs that are only distributed in binary form. Secondly, applying dynamic slicing at the binary level significantly eases deployment of our fuzzing systems by allowing seamless tracking of information flows through library code. We discuss this benefit of binary level dynamic slicing in more detail in section 4.3. Since the express goal of fuzzing is to trigger abnormal behavior in the program under test, it is also important that the slicer gracefully scales to longer-than-usual program runs when using dynamic slicing in a fuzzing pipeline, to avoid failure modes such as out-of-memory errors or extreme slowdowns.

1.7 Thesis Outline

The remainder of the thesis is structured as follows:

Chapter 2 provides some technical background information on static and dynamic analysis of binary code, in order to make the thesis more self-contained.

Chapter 3 surveys related work.

Chapter 4 summarizes the contributions outlined above in more detail.

Chapter 5 concludes the thesis and discusses future work.

1.8 List of Publications

Included Papers

Paper I: Ulf Kargén and Nahid Shahmehri. **Efficient Utilization of Secondary Storage for Scalable Dynamic Slicing.** In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 2014, pp. 155–164.

Paper II: Ulf Kargén and Nahid Shahmehri. **Towards Robust Instruction-Level Trace Alignment of Binary Code.** In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 2017, pp. 342–352.

Paper III: Ulf Kargén and Nahid Shahmehri. **Turning Programs against Each Other: High Coverage Fuzz-Testing using Binary-Code Mutation and Dynamic Slicing.** In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 782–792.

Paper IV: Ulf Kargén and Nahid Shahmehri. **Speeding Up Bug Finding using Focused Fuzzing.** In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. 2018, Article 7.

Other Publications

Ulf Kargén and Nahid Shahmehri. **Towards Accurate Binary Correspondence Using Runtime-Observed Values.** In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 438–442.

Ulf Kargén and Nahid Shahmehri. **InputTracer: A Data-Flow Analysis Tool for Manual Program Comprehension of x86 Binaries.** In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 2012, pp. 138–143.

Chapter 2

Background

In this chapter, we give a brief overview of fundamental techniques for binary code analysis. Even though the focus of our work has been on dynamic analysis, we also discuss static analysis of binary code, in order to highlight complementary strengths and limitations of both approaches.

While dynamic analysis only provides information on code that actually executes, static analysis can give insights into the workings of an entire binary program. Static analysis is also often more scalable than dynamic analysis, since the latter requires actual execution of the analyzed binary. Dynamic analysis generally also entails *instrumenting* the binary, i.e. somehow injecting extra code that records information about execution behavior. This instrumentation typically introduces a significant runtime slowdown. Another benefit of static analysis is that it is easier to automate, since dynamic analysis requires a working runtime environment for the binary. Persistent side-effects from executing the analyzed binary, e.g., file system changes, also need to be taken into consideration when using dynamic analysis. However, static analysis of binaries requires recovering high-level information, like control-flow structure, which has been lost upon compilation to machine code. This is often quite a challenging task, which frequently produces imprecise results. In the next two sections, we will elaborate on some of the respective challenges of static and dynamic binary code analysis.

2.1 Static Analysis of Binary Code

The first step in statically analyzing a binary executable is to *disassemble* the binary, i.e., to decode each machine code instruction into its corresponding assembler instruction. While this may seem like a trivial task at first glance, static disassembly is, in fact, an undecidable problem in the general case. While existing heuristic methods generally work well, there are several fairly common code constructs that cause state-of-the-art disassemblers to fail to produce correct or complete results.

Static analysis methods that perform any kind of global control or data flow analysis require a correct control flow graph (CFG). While it is fairly straightforward to extract a CFG from high-level program code with structured control-flow, correctly and completely reconstructing the control flow from a binary is a very challenging problem that often requires employing various compiler-specific heuristics.

Many static analysis methods also require the set of functions in the binary, along with their respective start and end addresses, to be known. Recovering this information from binaries can also be challenging, especially in highly optimized code.

Static Disassembly

While translating a single machine code instruction to get its opcode and operands is a trivial task, the main challenge when disassembling a binary is to discern code from data. Some compilers intersperse data with code in the code section of binaries that they produced [36]. *Linear disassemblers*, such as GNU objdump [45], make a single linear sweep over the binary. Therefore, any data in the code section will be incorrectly interpreted as instructions. Since modern instruction sets are typically quite dense, attempting to disassemble arbitrary data often results in valid instructions. Such false instructions can, in turn, reduce the accuracy of higher-level static analysis passes, such as CFG recovery. For architectures with variable-size instructions, such as x86, attempting to disassemble data will also often “desynchronize” the instruction stream, so that even valid instructions after the data will be incorrectly disassembled. However, x86 disassembly tends to automatically re-synchronize after a few instructions [77].

More advanced *recursive disassemblers*, on the other hand, recursively follow the control-flow of a binary, and thus avoid attempting to disassemble non-code regions. The main drawback of this approach is that all parts of the binary code may not be discovered and disassembled. The completeness of the disassembly is highly dependent on the ability of the disassembler to correctly reconstruct all control-flow edges. In practice, therefore, recursive disassemblers perform disassembly and CFG recovery simultaneously.

Code obfuscation, which is very common in malicious software, can also be used to confuse the disassembly process. A common technique is to “trick” a disassembler into disassembling junk data that has been inserted into the code section [77]. While linear disassemblers can be easily defeated simply by inserting junk data at various locations in the program code, it is also possible to trick recursive disassemblers. For example, one common obfuscation technique is *bogus control flow*, which works by inserting conditional branch instructions where only one branch outcome is realizable during runtime, and the other branch direction points to junk data. Since the disassembler will generally assume that both outcomes are possible when analyzing conditional

branches, it will follow the non-realizable branch direction at some point and try to disassemble the junk data. Another obfuscation trick, which is also occasionally observed in non-obfuscated benign software, is overlapping instructions [80].

Control Flow Graph Recovery

In non-obfuscated binaries, the destination addresses of most branch instructions are *immediate*, i.e. hard-coded. Therefore, it is typically straightforward to extract a partial CFG by recursively tracking the targets of branches. However, *indirect branches*, i.e., branches where the target address is computed during runtime, are a major hurdle for CFG recovery. Two major sources of indirect branches are jump-tables and function calls via pointers. Jump tables are commonly used by compilers to efficiently implement switch-case constructs. Consider, for example, the following C-code:

```
switch(i) {
    case 0:  ... break;
    case 1:  ... break;
    case 2:  ... break;
    default: ... break;
}
```

An efficient way to generate machine code for the above is to create a jump-table with addresses to code for handling each case, and then emitting code that indexes into this jump table based on the value of *i*. For example, assuming 32-bit (4 byte) pointers, the compiler will emit code equivalent to `jump to [TableBase + 4*i]`. (Of course, an additional check that $0 \leq i \leq 2$ is also needed.) Since the precise details of how jump tables are used differ among compilers, heuristic methods are needed to handle jump tables when constructing the CFG from a binary.

In a recent study by Andriesse et al. [4], 9 different disassemblers were tested against x86 binaries produced with three different compilers. While the state-of-the-art commercial disassembler IDA Pro achieved close to perfect CFG recovery for all three compilers, the other tools achieved markedly lower accuracy. Along with the commercial tool Hopper [54], the open-source binary analysis framework Dyninst [11] achieved about 90% accurate recovery of basic blocks. The other tools, which were all open-source, achieved around 75% basic block recovery. The accuracy of CFG recovery also differed noticeably across different tools, depending on the compiler used to create the binary, with some tools performing better on code generated by a certain compiler. The exception was IDA Pro, which performed consistently well regardless of compiler. Similarly, all tools except IDA Pro showed less accurate results for, respectively, optimized code compared to unoptimized, C++ programs compared to C programs, and 64-bit binaries compared to 32-bit binaries.

While the reasons for the observed differences are not discussed in detail in the study, one likely explanation for why IDA Pro outperforms other tools is that, being a commercial solution, its authors have more time and resources at their disposal to craft finely-tuned heuristics for different compilers.

CFG recovery can also be effectively prevented using obfuscation. A common technique is to replace regular branch instructions with indirect branches, whose targets cannot easily be determined statically [118].

Function Boundary Identification

Function identification in binaries is a twofold problem. The first part of the problem is to identify all functions present in the binary. Functions called with immediate addressing can be identified during CFG reconstruction simply by recording the targets of all function-call instructions that are encountered. Identification of functions that are only called with indirect addressing is more challenging, however, as such functions must be identified heuristically. A common technique is to try to detect function *prologues* (i.e. code for initializing the function’s stack frame) using pre-determined signatures. Signatures can either be manually crafted or learned using, e.g., neural networks [8]. Another, more simple heuristic used by some tools is to simply consider code blocks that are not reachable in the reconstructed CFG as the start of a function. In the aforementioned study by Andriesse et al., function identification was shown to be a challenging area for binary analysis systems. For example, the state-of-the-art commercial tool IDA Pro only identified 60–70% of function starts. The vast majority of missed functions were due to indirect addressing. Some of the other tools performed better at function identification due to using the heuristics mentioned above, but those tools also had high false-positive rates (around 10–20%).

The other main challenge of function identification is to locate the physical boundaries of the function’s code within the binary. A common optimization that complicates function boundary identification is *tail calls*. If the compiler can statically determine that one function is only ever called at the very end of another function, it can replace the call with an unconditional branch, in order to avoid the overhead of performing a complete function call. Tail-call optimizations may cause a binary analysis tool to mistakenly determine that the code of both the caller and callee belong to the same function. Another challenging case for function boundary identification is when functions are laid out non-contiguously in memory in order to improve cache performance [80].

2.2 Dynamic Analysis of Binary Code

While static analysis of binary code will always entail a certain degree of inaccuracy due to reliance on heuristic methods, dynamic analysis has the

potential to deliver precise information about program behavior. Statically unknown information, such as indirect branches or function calls, poses no problem to dynamic analysis, since information on branch targets is available during runtime. Static obfuscation, such as bogus control flow or obfuscation of branch targets, is not a problem for dynamic analysis either, as the analysis engine sees exactly the same instruction stream as the CPU itself. Dynamic analysis can also overcome *packing*, which is a very common obfuscation method used in modern malware. Packing works by compressing and/or encrypting the original binary executable and wrapping it inside another executable, which only serves the purpose of extracting and running the original binary code when executed.

Dynamic analysis requires some degree of instrumentation, i.e. intermixing the execution of the target program's code with additional analysis code that records information on program behavior. Therefore, dynamic analysis always leads to a certain amount of overhead, with more granular analysis generally leading to greater slowdowns. Many dynamic analysis methods have been proposed in the literature, targeting different points along the granularity/overhead tradeoff spectrum. Methods also differ in how instrumentation is applied. In the remainder of this section, we first discuss some common methods for dynamic analysis. We then discuss some specific techniques that are commonly used to implement dynamic information flow tracking. Finally, it should be noted that, just like obfuscation can hinder static analysis, several methods for evading dynamic analysis are known to exist. For completeness, we conclude this section with a brief discussion of methods for dynamic analysis evasion.

Dynamic Analysis Methods

Debuggers are ubiquitous tools for manual dynamic analysis. Scriptable debuggers can also be used to implement simple automated dynamic analysis. For example, a programmable debugger can be used to single-step through a program execution and apply some kind of analysis at each instruction. Debuggers typically rely on hardware features of the CPU to single-step through code or set breakpoints. At each single-step (or when encountering a breakpoint), the CPU will trap into the OS kernel, which in turn will transfer control to the debugger via a debugging API. This frequent context-switching incurs very high overhead, making debuggers less well-suited for fine-grained dynamic analysis.

API hooking is a more scalable approach to dynamic analysis that works by "hooking" function calls, typically API functions. A common hooking technique entails overwriting the entry point of the function of interest with an unconditional branch to analysis code¹. When the function is called, execution

¹Note that the code is only overwritten in the virtual memory space of the process being analyzed, not in the executable file on disk.

is redirected to the analysis code, which can record information about the call. The analysis code then redirects execution back to the hooked function, so that it can execute normally. Since analysis is only carried out at the granularity of (selected) function calls, hooking incurs low overhead, and is therefore suitable for large-scale automated dynamic analysis. For example, many malware analysis frameworks utilize hooking to fingerprint the behavior of malicious software [34].

Dynamic binary instrumentation (DBI) allows for implementing very fine-grained dynamic analysis. DBI frameworks provide an API that tool developers can use to apply instrumentation at the basic block or instruction level. When DBI is applied to a binary, the original binary code never executes directly. Instead, the DBI framework maintains a cache of instrumented code blocks. When a new (i.e. previously unseen) piece of code is about to execute, an API callback is invoked, allowing the DBI tool to inspect the code and apply instrumentation. The code block is then "recompiled", interspersing the added analysis code with the original code, and the recompiled block is added to the code cache and executed. On any subsequent execution of the same block, the instrumented version in the code cache will execute, rather than the original code. The DBI framework maintains control at all times by making sure that execution is always returned to itself after an instrumented code block finishes executing. Code blocks in the cache are sometimes referred to as *dynamic basic blocks*. Note that a dynamic basic block is not a "real" basic block, but simply a linear stretch of code typically ending in a branch instruction, since determining the set of "real" basic blocks requires static CFG reconstruction. Each non-fallthrough control-transfer results in the creation of a new dynamic basic block in the code cache. Therefore, there may be several partially overlapping dynamic basic blocks in the cache, if there are multiple branches into the same single-exit stretch of code. Figure 2.1 illustrates this concept. The first time the program branches to the code at line 1, the DBI system will disassemble the code up until the branch at line 5 and invoke a callback that inspects the newly-discovered block and inserts instrumentation code. The instrumented block at lines 1–5 (*A* in the figure) is finally added to the code cache, from which it is executed. If the branch at line 5 is not taken, the above procedure is repeated for lines 6–9, creating the block *B* in the code cache. When the program again branches to line 1 at some later point, the instrumented block *A* is directly executed from the code cache. However, if this time around the branch to line 7 is taken, the procedure for instrumenting a block and adding it to the code cache is now invoked for the block 7–9, resulting in the addition of block *C* to the cache. For future invocations of the code, the DBI system will fetch the appropriate block (*B* or *C*) from the code cache, depending on the branch decision at line 5.

DBI applies instrumentation to all (executed) code within the analyzed program's virtual memory space, including all shared libraries. Only user

Figure 2.1: Pseudo-assembly listing demonstrating the concept of dynamic basic blocks in DBI systems.

mode code can be instrumented, however. Therefore, DBI frameworks have an API to allow interception and inspection of all system calls, which allows tools to monitor interactions with the OS.

The overhead of dynamic analysis with DBI has three main sources: Applying the initial instrumentation to every (executed) code block, execution of the analysis code, and additional context-switching overhead between analysis code and original code. The overhead of adding instrumentation to a block is amortized over every execution of that block. Therefore, this source of overhead is often negligible when a large part of the execution takes place inside a (relatively) small number of loops. However, if a large part of the execution time is due to code that only runs once, the instrumentation overhead is very significant. This is often a problem for, e.g., programs with a GUI, since a large part of the execution time is spent on one-time initialization of the GUI subsystem prior to executing the core program logic.

Since analysis code is directly mixed with original code, care must be taken that side-effects of analysis code do not interfere with the semantics of the original program. The context-switching overhead from DBI stems from, e.g., register spilling and saving/restoring the condition-flags register to avoid such side effects. Existing DBI frameworks differ in terms of how much support they offer tool writers in handling context-switching. For example, when using Pin [78], all analysis code is written as normal C++ functions, and Pin takes care of adding context-switching code during instrumentation. DynamoRIO [18], on the other hand, requires all analysis code to be provided in assembly form, and properly handling side effects is left entirely to the tool writer. DBI frameworks also differ in their level of abstraction of the analyzed binary code. While frameworks like DynamoRIO and Pin only provide tool writers with an interface to analyze and instrument x86 binary code, Valgrind [91] supports many different CPU architectures, and all binary code is translated into a common intermediate representation (IR). Inspection and instrumentation is carried out on the IR, which is subsequently recompiled into native code and put into the code cache. This simplifies tool writing, since developers need

not care about the intricacies of different machine code languages, but comes at the cost of higher overhead: The baseline overhead, i.e. the overhead of running a program through a minimal DBI tool without any analysis code, is on average about 6x for Valgrind, while it is about 1.5x for DynamoRIO and Pin [78].

Static instrumentation is an alternative to DBI, if the same binary is to be executed many times with the same instrumentation, since the cost of re-applying instrumentation on every execution can be avoided. Static binary instrumentation works by directly patching the machine code of an executable file to insert analysis code. Since safely applying binary patching requires accurate CFG recovery [11], static instrumentation is limited by the difficulties of static analysis of binary code that were discussed in section 2.1. Therefore, static instrumentation may, for example, be difficult for obfuscated code [103].

Emulation allows for stricter isolation of analyzed code. For example, many antivirus products execute suspicious binaries in an emulator to detect malicious runtime behavior and enable analysis of packed code [70]. To prevent damage to the host system, the analyzed binary runs on a simulated CPU, and all interactions with the OS (i.e. system calls) are also simulated. Emulation is also used by several academic binary-analysis systems. For example, angr [108] lifts binary code for various CPU architectures to an IR, which can be executed inside an emulator to implement, e.g., *concolic execution* (see section 3.2). KLEE [19] similarly implements dynamic symbolic execution using emulation of LLVM bytecode.

One limitation of emulators that simulate interactions with the OS is that, inevitably, such simulations can never perfectly mimic a real OS. Small discrepancies between real and simulated behavior can lead to inaccurate analysis results. *Full-system emulators* avoid this problem by running an off-the-shelf operating system inside a hardware emulator. Dynamic analysis can then be carried out by extending the emulator to monitor code executed by the simulated CPU. For example, several academic malware-analysis platforms use full-system emulation [125, 32]. Apart from eliminating emulation inaccuracies, a significant benefit of this approach is that interactions with the OS and other processes can be directly observed and analyzed. However, a challenge with full-system emulation is the *semantic gap* between the behavior of programs running inside the emulator and the observable effects of this behavior from the perspective of the emulator itself. The emulator only sees a stream of instructions and operands being processed by the simulated CPU. Typically, only parts of the instruction stream belong to processes or OS components that are of interest for the analysis, but the separation of executing code into, e.g., processes or threads is not visible outside the emulated system itself. Therefore, dynamic analysis using full-system emulation typically requires some degree of *virtual-machine introspection*, i.e. mechanisms to “peek” inside the emulated system. For example, TEMU, the dynamic-analysis com-

ponent of the BitBlaze binary analysis platform [111], uses a custom driver installed in the guest OS to provide information on, e.g., which process is currently executing on the simulated CPU.

Dynamic Information Flow Tracking

A common application of fine-grained dynamic binary analysis is *information flow tracking*, such as *taint analysis* (see Section 3.1). The flow of information is dynamically traced throughout execution by observing computations. Often, it is interesting to observe the flow of information from an *information source* to an *information sink*. Commonly, points in the execution where the program reads inputs are considered information sources, while points when data is written to output is considered information sinks. For example, information flow tracking can be used to detect privacy leaks by tracing the flow of information read from privacy-sensitive data sources [131]. If the sensitive information flows into a sink that, e.g., transmits data to the Internet, the program can be flagged as one that leaks private information.

To detect information flows from sources to sinks, it is necessary to monitor the analyzed program's interactions with the operating system and possibly other processes. Most commonly, programs interact with the outside world through system calls. By instrumenting system calls and performing proper bookkeeping, it is possible for the analysis tool to monitor when the program opens, e.g., files or network sockets, and when data is read/written to/from a socket or file.

Tracking information flow also requires maintaining a *shadow state* [90]. Every register has a shadow, which contains metadata about the corresponding real register. Similarly, a shadow is maintained for each memory location. Every instruction that operates on registers or memory is also instrumented to propagate metadata from/to the corresponding shadow locations. For example, to detect privacy leaks, an analysis system would first monitor system calls to detect when sensitive data is read into memory. The corresponding range of memory address in the shadow memory is then marked as "tainted". If an instruction, e.g., reads data from memory into register R , the taint status will be copied from the corresponding part of shadow memory to the shadow of R . In this way, the transitive propagation of tainted data can be tracked during execution.

To implement a shadow state, every thread must have its own set of shadow registers, and thread switches must be monitored. One challenge when employing a shadow state to analyze multi-threaded programs is to avoid data races when updating the shadow memory concurrently in different threads. One pragmatic solution, used by, e.g., the popular Memcheck tool [106], is to simply serialize execution.

Subverting Dynamic Analysis

Just like static analysis can be complicated or even completely prevented by using static obfuscation, dynamic analysis can be subverted in various ways. While dynamic analysis of malicious or obfuscated code has not been the express goal of the work presented in this thesis, for completeness we nonetheless give a brief overview of anti-dynamic analysis methods.

Most mechanisms for preventing dynamic analysis rely on the target program being able to detect that it is being analyzed and change its behavior accordingly. For example, a program can exit prematurely to hinder reverse engineering, or a piece of malware can refrain from exhibiting malicious behavior to avoid detection.

Most operating systems provide an API that allows programs to query whether they are being debugged. This constitutes a simple way for programs to detect if they are being subjected to dynamic analysis with a debugger. For other dynamic binary analysis techniques, more sophisticated methods are needed. Such methods either rely on detecting specific artifacts of the analysis environment, or observable indirect effects of the analysis. As an example of the former, it is often possible for a program to tell that it is being executed within a virtual machine or emulator by detecting, e.g., virtual hardware devices with specific names or identifiers [24]. Similarly, runtime analysis environments like DBI systems can often be detected by scanning the virtual memory space to find memory objects used by the DBI framework [97]. One of the most common indirect methods for detecting dynamic analysis is timing measurements. Since the analysis typically incurs significant overhead, a program can measure the time taken to execute a certain operation, and, e.g., terminate if the operation takes significantly longer than what is expected during normal execution. A more sophisticated method, which can be used to detect virtual environments or emulators, is to detect subtle differences in the semantics of certain instructions compared to real hardware [94, 107].

An interesting possibility when performing dynamic analysis is for the analysis system to reactively try to counter anti-analysis attempts from the target program. Since the target program typically runs under the "supervision" of the analysis environment, it is possible to recognize attempts at detecting the analysis environment, and then manipulate the computations of the target program to foil such attempts [97].

Chapter 3

Related Work

In this section we survey work related to the research problems addressed in this thesis. We begin with a discussion of dynamic slicing and related dynamic information flow analysis methods. We then discuss classical approaches, as well as current research directions, in the field of fuzzing. Lastly, we survey related work in the field of trace alignment, and the broader field of binary code matching.

3.1 Dynamic Information Flow Analysis

In this section, we first discuss previous work on dynamic slicing and approaches for addressing the scalability challenges associated with the technique. We also survey previous work on *dynamic taint analysis*, which is a related but more coarse-grained form of dynamic information flow analysis that has become particularly popular for analyzing binary programs. Since the aim of our work has been analysis of binary code, we focus our discussion of information flow analysis on dynamic approaches. While static information flow analysis methods, such as static slicing, have been both thoroughly studied in the literature and have also been used in practice, applying them to machine code has proven very challenging due to the limited accuracy of static binary code analysis. Cifuentes et al. proposed a method for static slicing of binaries [27], but their method is restricted to *intraprocedural* slices, i.e. slices within a specific function. Kiss et al. attempted to compute global (interprocedural) static slices on binaries, but found that the accuracy was too low for practical applications, as slices frequently consisted of half of the instructions in the entire binary [66]. In a follow-up work, the same authors discovered that a large contributing factor to the inaccuracy was statically unresolved indirect function calls. By augmenting their method with dynamic information on call targets, they were able to increase the accuracy, but found that slices still covered about 20% of the entire binary [67].

Dynamic Slicing

Dynamic slicing was first proposed in 1988 by Korel and Laski [69]. Their method was based on solving data flow equations to compute backward slices, and was imprecise in the sense that the computed slices might be unnecessarily large (i.e. they might contain statements not actually affecting the slicing criterion). Agrawal and Horgan [2] introduced four algorithms for computing backward dynamic slices from a directed acyclic graph called the *program dependence graph* (PDG), where statements are represented by nodes, and dependencies between statements are represented by directed edges. Once constructed, the PDG can be used to compute slices by finding all nodes that are reachable from the node corresponding to the slicing criterion. The first two algorithms by Agrawal and Horgan sacrificed accuracy for speed, and computed imprecise slices (with their second algorithm being equivalent to the approach by Korel and Laski). The third algorithm used a variant of the PDG in which each executed *instance* of a statement would be represented by a node. The resulting graph, called the *dynamic dependence graph*, could be used to compute precise slices. However, Algorithm 3 has high storage cost, since the size of the dynamic dependence graph will be proportional to the total number of executed program statements. Their fourth algorithm effectively precomputes all possible slices, and is thus very computationally expensive.

A PDG contains edges for both *data dependencies* and *control dependencies*. A statement A is data dependent on another statement B if one of A 's input operands was *defined* (i.e. produced) by B . A is intuitively control dependent on statement C if C controls whether A is executed or not. In contrast to static dependence analysis, which aims to discover all possible dependencies among statements, dynamic dependence analysis records dependencies that have actually been exhibited for every statement instance during execution. While early works used slightly different notions of dynamic control dependence, Xin and Zhang [129] formalized the concept based on the definition of static control dependence by Ferrante et al. [40]. Since finding dynamic control dependencies requires static control dependence information, static CFG recovery is required in the context of binary code analysis.

Figure 3.1 shows a small C function for computing exponentiation, and the dynamic dependence graph when the function is executed on inputs `base = 2` and `exp = 3`. Nodes are labeled with corresponding line numbers in the code listing. Solid edges denote data dependencies and dashed edges denote control dependencies. Note that, since the loop at lines 6–7 is executed for two iterations, there are two sets of nodes labeled 6 and 7. The edge between the nodes labeled 6 denotes the loop-carried dependence for variable `i`, similarly to the edge between the nodes labeled 7 denoting the loop-carried dependence for variable `result`. Computing a backward slice on the return value at line 12 simply entails finding all nodes reachable from node 12, yielding the slice

Figure 3.1: A simple function for computing exponentiation and its dynamic dependence graph for inputs **base** = 2 and **exp** = 3.

$\{1, 1', 2, 3, 5, 6, 7, 9, 12\}$. If only data-dependencies are considered, the slice would instead be $\{1, 5, 7, 12\}$. A forward slice is computed by traversing edges in the opposite direction. For example, computing a forward slice on the input **exp** would yield $\{1', 2, 3, 6, 7, 9, 12\}$, or $\{1', 2, 6, 9\}$ if only data dependencies are considered.

Dynamic slicing was quickly recognized as a powerful aid for various program comprehension tasks, such as debugging [1]. Since its introduction in the late 1980s, numerous works have utilized various variants of dynamic slicing to aid in debugging. (See for example Wong et al. [126] for a survey of some more recent works that use dynamic slicing for fault localization.) The utility of dynamic slicing for debugging has also been empirically demonstrated [139, 135]. However, one of the main impediments to the widespread use of dynamic slicing has been scalability. Because of the high storage cost of Agrawal and Horgan’s precise algorithm, implementations that store the graph in RAM tend to run out of memory quickly when used in practice. Several works attempt to solve this problem by various time-versus-space trade-offs, which sacrifice slice computation speed in order to reduce the memory footprint of their algorithms. Zhang et al. [136] proposed computing slices directly from an instruction trace instead of first building the dynamic dependence graph from the trace and then traversing the graph to compute slices. Since it is very time-consuming to traverse over the entire trace to recover data and control dependencies that are relevant to a particular slice, they also proposed a method for speeding up slice computation by annotating the trace at fixed intervals with information about all downward-exposed variable definitions. This allows quickly skipping past parts of the trace that are not relevant to a particular slice computation. In their experiments, Zhang et al. compared their trace-based method with Agrawal and Horgan’s traditional graph-based

one. Due to its large memory requirements, the graph-based method would run out of memory in the majority of cases. However, for cases where the graph could fit in memory, the graph-based method outperformed the trace-based one by several orders of magnitude in terms of slice computation speed.

Even though a trace is a more compact representation of an execution than a dynamic dependence graph, traces that capture long program executions can still become very large. Wang et al. [120] addressed this problem in the context of slicing Java programs. They proposed a method for efficiently compressing Java bytecode traces, along with a corresponding algorithm to compute slices directly from compressed traces without requiring decompression.

As shown by, e.g., the experiments by Zhang et al. [136], trace-based methods have high overhead compared to graph-based methods. In particular, in cases where multiple slices need to be computed for the same execution, much time is wasted on re-computing dependencies from the trace for each slice, instead of starting by resolving all dependencies in a graph before constructing slices. Zhang and Gupta [137] proposed a method for compressing the dynamic dependence graph sufficiently to fit in RAM, using several techniques utilizing both static and dynamic analysis. While their method is faster at computing slices than trace-based approaches, it requires a time-consuming pre-processing stage. Since their method works on C source code and requires static information that cannot be accurately recovered from binaries, several of the techniques for compacting the graph is not applicable to the binary code case. Nagarajan et al. [89] describe a dynamic slicing system that is specifically tailored for debugging, which works directly at the binary level. Their system uses a subset of the compaction techniques proposed by Zhang and Gupta that can be applied on the binary level. To address scalability, their system only stores dependencies for a small window of recently-executed instructions. This design decision is based on the observation that the faulty computation that causes a software failure (e.g., a crash) often happens close in time to the actual failure.

Dynamic Taint Analysis

Due to the scalability challenges of dynamic slicing, more coarse-grained but faster data flow analysis methods have been proposed. *Dynamic taint analysis* (DTA) is a relatively "lightweight" data flow analysis technique that is also readily applicable to binary code. For this reason, it has been applied in a wide variety of fields, such as malware analysis [9, 131, 98], automatic detection of software exploitation attempts [92, 28, 14, 100], detection of privacy leaks [140, 37] and fuzzing [43, 121, 50, 101].

Dynamic taint analysis is a forward data flow analysis method in which variables of interest, e.g., certain program inputs, are marked as "tainted". The taint status of data is then propagated as the program executes by means of a shadow state, so that new data that is derived from a tainted variable also

<pre> . . a = read_one_value(file_name) b = a * 2 c = a + b print(c) . . </pre>	<pre> . . a = read_one_value(file_name) shadow_a = check_and_taint(file_name) b = a * 2 shadow_b = shadow_a c = a + b shadow_c = union(shadow_a, shadow_b) print(c) taint_sink(shadow_c) . . </pre>
(a)	(b)

Figure 3.2: Pseudo-code example of instrumentation added to a program for performing DTA. (a) shows the original version of the code and (b) shows the instrumented version.

becomes tainted. Certain points of interest in the program can be defined as "taint sinks", so that a notification is generated if tainted data reaches such a point. For example, when using dynamic taint analysis to detect leaking of private data, input from privacy-sensitive sources is marked as tainted, while, e.g., networking API functions can be defined as taint sinks. In this way, it is possible to detect if the program leaks private information over the Internet.

Figure 3.2 shows a simple example of how instrumentation for DTA is added to a program. The code in (a) is instrumented by inserting extra statements for propagating taint through a shadow state. The added instrumentation code is highlighted in gray in (b). Each variable is assigned a shadow-variable to hold its taint status. Note that, for clarity of exposition, we show instrumentation on the source-code level in this example. When performing DTA on binary code, taint is propagated at the level of individual registers and memory addresses, and we use shadow memory and shadow registers instead of shadow variables. At the point in the code where a value is read from file, instrumentation is inserted to check if `file_name` is registered as a taint source, and the value of `shadow_a` is set accordingly. At the next line, the taint status from `shadow_a` is simply copied to `shadow_b`, as arithmetic with a constant does not change the taint status of a variable. When computing the sum of two variables at the consecutive line, however, the taint status of `c` is set to the union of `a`'s and `b`'s taint status¹. Finally, since `print` is considered a taint sink in this example, `taint_sink` is called on `shadow_c` to check if tainted data has flowed into a sink.

¹Note that in this case we could have statically deduced that the taint status of `a` and `b` is the same, thereby avoiding the union operation. However, in a typical DTA implementation taint propagation instrumentation is simply added "mechanically" based on the semantics of individual statements or instructions.

While less expensive than dynamic slicing, binary-level DTA still typically incurs an overhead of 5-10x [64]. DTA systems that are tailored to specific use cases can have lower overheads. For example, Minemu by Bosman et al. [14] is tailored for on-line detection of software exploits on production systems using DTA, and achieves overheads as low as 1.5x, but it is restricted by design to only work on 32-bit x86 executables.

Compared to dynamic slicing, DTA has two primary limitations. First, DTA can reveal that tainted data has reached a taint sink, but provides no information on *how* this happened (e.g., which instructions were involved). Second, taint is typically treated as a binary property, i.e. a variable is either tainted or not. This means that DTA can only be used to detect *coarse-grained* information flows, like, for example, if *any* part of a program input ended up influencing a taint sink. On the other hand, dynamic slicing can reveal exactly which instructions were influenced by, e.g., a specific byte of an input. Some DTA implementations allow slightly more granular data flow analysis by using a shadow state in which taint can be represented using one or more entire bytes (instead of a single bit). Each bit position can then represent a specific "color" (e.g., 8 distinct "colors" if one byte is used to hold taint information). An example of such an implementation of DTA is libdft by Kemerlis et al. [64].

The first limitation of DTA can be addressed by doing "bookkeeping" at every instruction execution² to achieve data flow tracking similar to forward dynamic slicing. (Note that backwards data flow analysis, like backwards dynamic slicing, cannot be achieved using DTA.) The second limitation can be overcome by using *multi-label tainting*, where taint is represented as a *set* of taint labels, instead of only one bit (or a few bits). Each byte of input can then be represented by a unique taint label. While several such systems exist [43, 121, 63, 50, 112, 101], they tend to have high memory overhead because each element of the shadow state may need to hold a large set of values. The computational overhead is also large, because taint propagation entails computing the union of taint label sets, instead of performing simple bitwise logic operations to propagate taint.

A challenge that is common to all dynamic information flow analyses is how to handle *indirect information flows*. In contrast to direct data flows, where the value of one variable is directly derived from other variables using, e.g., arithmetic instructions, indirect flows often represent a more "loose" coupling between variables. Two types of indirect flows are often considered in dynamic information flow analyses. The first type is due to *address dependencies*, which stem from using one variable to compute the address of another variable. For example, if a variable is used as an index to fetch a value from an array and store it in variable *x*, then there is an address dependency, and thus an indirect information flow, between the index variable and *x*. The other major type of

²That is, every instruction is treated as a taint sink

indirect information flow is due to control-dependent variable assignments. For example, in the code snippet `if a<0 then b=1 else b=2`, the variable `b` is control-dependent on the variable `a`, and therefore there is an indirect information flow from `a` to `b`.

When performing DTA on binary programs, a well-known problem is that considering indirect flows when propagating taint often leads to a kind of "taint explosion", i.e., that almost all variables end up as tainted due to spurious information flows. This problem is sometimes referred to as *overtainting*. However, not considering indirect flows can lead to *undertainting*, i.e. that important information flows are missed. For example, not considering address dependencies can lead to missing information flows through lookup tables. Prominent examples of such cases are the `toupper` and `tolower` C standard library functions, which translate an ASCII character to its upper- or lowercase variant, respectively, by using a translation table. Similarly, not considering control dependencies can also lead to undertainting.

Due to problems with overtainting, most existing DTA implementations only consider direct data dependencies. However, several works attempt to address the problem of indirect information flows for DTA. Slowinska and Bos [109] discussed several heuristics for dealing with overtainting due to address dependencies, but found that no existing heuristic delivered satisfactory results, and concluded that tracking address dependencies on x86 binary code was in many cases close to pointless, due to problems with taint explosion. Several authors have also noted that considering control dependencies in DTA creates similar (or worse) problems with taint explosion. Bao et al. [7] defined the notion of *strict control dependence* to deal with spurious information flows from control dependencies. The main idea is to only consider control dependencies that stem from branch predicates where the tainted variable is checked for *equivalence* with another variable or constant, as such code constructs represent a strong correlation between the tainted variable and control-dependent assignments. Kang et al. proposed a different technique to address the same problem [61]. During an offline analysis phase, they first identify a small subset of all control dependencies that must be considered for taint propagation to avoid undertainting. By only propagating indirect taint through these control dependencies during online DTA, they reduce the probability of overtainting.

3.2 Fuzzing

The term "fuzzing" was coined by Barton P. Miller after an incident in 1987. Miller was working over a dialup connection to a mainframe during a thunderstorm, and observed that "fuzz on the line" introduced by the thunderstorm, (i.e. random bit-errors in program inputs,) would often crash common UNIX utility programs. This inspired Miller et al. to empirically study the robust-

ness of UNIX command-line utilities using random input generation [82, 83]. Today, fuzzing is considered an invaluable aid for discovering bugs and improving the security and robustness of software. In this section, we discuss the three basic fuzzing paradigms — black, white, and greybox fuzzing — in more detail.

Blackbox Fuzzing

Blackbox fuzzing, using either mutational or generational input generation strategies, became popular among professional penetration testers, and with the hacker community at large, during the early 2000’s. Mutational fuzzers [52, 93], which (much like Miller’s thunderstorm) work by introducing random errors into valid seed inputs, proved to be surprisingly effective at finding memory corruption bugs in popular software [114]. However, mutational fuzzers have many severe limitations. First, they can only do a very narrow exploration of the input space “close” to the provided seed inputs. Therefore, a good selection of diverse seeds is very important when using mutational fuzzing. For example, Rebert et al. showed that the seed selection strategy can have a significant influence on the number of bugs found during fuzzing [102]. Another limitation of mutational fuzzers is that their effectiveness is often highly dependent on various tunable parameters. A crucial parameter is the *mutation ratio*, which determines how large a fraction of the bits in a seed are mutated to generate an input. A mutation ratio that is too high risks irrecoverably breaking the structure of the input, causing it to be rejected by early sanity checks in the PUT (program under test). With a mutation ratio that is too low, inputs may instead never become sufficiently malformed to trigger any bugs. Since the optimal mutation ratio differs significantly among different input formats, the effectiveness of a mutational fuzzer is often largely dependent on its user’s ability to tune this ratio, based on experience and intuition. Householder and Foote [56] attempted to address this shortcoming by using a machine learning approach based on crash rate feedback to automatically tune various fuzzing parameters. Cha et al. [21] proposed a more advanced method, using dynamic data flow analysis to determine dependencies between input bytes, and then using this information to compute the optimal mutation ratio.

While mutational fuzzers became popular due to their simplicity, their effectiveness for finding bugs is fundamentally limited by their lack of semantic insight. For this reason, they are mostly suited to finding relatively “shallow” bugs in programs with fairly unstructured input formats. For programs with rigid requirements on the structure of their inputs (e.g., XML parsers, compilers, etc.) a generational approach is often required to achieve reasonable code coverage with black-box fuzzing. Generational fuzzers use a machine-readable specification (e.g., a grammar) to generate inputs that can pass complex syntactic checks. However, if all inputs adhered perfectly to the

format specification, the fuzzer would be useless for testing the robustness of the PUT to malformed inputs. Therefore, generational fuzzers typically use a number of *fuzzing heuristics* — inputs or input patterns that are known to have triggered bugs in the past [114]. Generational fuzzing frameworks like SPIKE [3] or Peach [95] provide a number of such fuzzing heuristics, such as using “extreme”³ values in parts of the input where an integer is expected. More advanced fuzzing heuristics can also be used. Holler et al. [53] used a generational fuzzing approach with code fragments known to have triggered bugs in the past to discover vulnerabilities in JavaScript and PHP interpreters. Compared to other fuzzing approaches, generational fuzzing is particularly suitable for fuzzing compilers and interpreters, due to the highly-structured nature of inputs and the availability of formal input specifications. Aside from Holler et al., other examples of successful programming-language fuzzers using a generational approach are the works by Yang et al. [130] and Veggiam et al. [117].

Whitebox Fuzzing

While generational fuzzing can achieve superior code coverage compared to random input mutation, crafting formal input specifications for fuzzing requires significant effort. For some particularly complex⁴ input formats, the amount of effort required has the potential to make generational fuzzing infeasible. Due to the limitations of black-box fuzzing, the research community started to shift its focus towards whitebox methods during the late 2000s. The development of practical whitebox fuzzers was made possible in large part due to advancements in SMT (satisfiability modulo theory) solving methods, and the availability of efficient SMT solvers like STP [42] and Z3 [31]. Whitebox fuzzers employ *symbolic execution* to discover bugs in programs. The program is executed in a (conceptual) symbolic machine, where inputs are treated as symbolic variables, and where each instruction execution results in a symbolic expression on input variables, rather than a computed value. (Typically, each byte of, e.g., an input file is considered a symbolic variable.) Each branch instruction executed in the symbolic machine adds a constraint on the inputs. By negating the constraint corresponding to a specific branch point and querying an SMT solver for a solution to the collection of constraints, a new set of input variables is obtained. The new input is guaranteed to take the execution down a different path, after the given branch point. (Alternatively, the solver will report that the constraint is unsatisfiable, meaning that the corresponding execution path is not possible.) Two approaches to symbolic execution exist. In systems like KLEE [19] and S2E [25], the symbolic execu-

³Very large positive or negative numbers, negative numbers where only positive are expected, etc.

⁴As an example, the document describing the Adobe PDF standard [58] has over 1300 pages.

tion will fork whenever a branch instruction is encountered. If querying the SMT solver shows that both directions of a branch are possible, two divergent “snapshots” of the symbolic state will be created. A downside of this approach is that a very large amount of memory will be required for storing symbolic states when testing complex software with deep program paths. A more memory-efficient approach is to use offline symbolic execution [105, 46]. With this approach, the program is executed from start to finish on each input, while recording a program trace. The trace is then processed symbolically to create constraints corresponding to the concrete execution, and the SMT solver is used to generate an input that takes a different path at some branch point. This approach reduces the memory footprint at the expense of taking longer to generate constraints, since the symbolic expressions need to be entirely recreated from scratch for each new execution.

The main impediments to the widespread use of whitebox methods are scalability and to some extent the high implementation complexity. The scalability problem is twofold. First, since the number of paths (and thus symbolic states) is exponential in the number of branches, whitebox methods suffer from *path explosion*. While symbolic execution can, in theory, achieve 100% code coverage by systematically exploring all different branch outcomes, this is only feasible in practice on very small programs. For example, Kuznetsov et al. [71] demonstrated that they could reach up to 90% basic block coverage when testing device drivers. For large and complex software it is necessary to use approaches that, similar to methods based on random input mutation, employ a more “sparse” exploration of the program state space. Examples of this approach include the systems SAGE [46] and Mayhem [20], which do not aim for exhaustive exploration of tested programs, but rather use search heuristics that prioritize exploring paths that have contributed the most to increasing code coverage. This facilitates a testing strategy that emphasizes finding as many bugs as possible given a fixed time budget, similar to classical blackbox fuzzers. Another way to combat path explosion is *hybrid fuzzing*, which combines greybox fuzzing with symbolic execution. We describe hybrid fuzzing in more detail below, in conjunction with the discussion of greybox fuzzing.

A more fundamental limitation of symbolic execution is that solving SMT queries is an NP-complete problem. Therefore, the time to solve path constraints will increase exponentially with the depth (i.e. number of branches) of the path. This often limits the depth of exploration that can be attained with a symbolic execution framework. For example, Bounimova et al. [16] mention that, in order to make SAGE scale to production use, deep program paths must be pruned in order to avoid spending excessive time on solving path constraints. This means that bugs that reside at deeper levels in the program control-flow structure may not be found.

Apart from fundamental scalability challenges, whitebox fuzzers are also challenging to implement due to their inherent complexity. When designing

systems based on symbolic execution, the question of how to handle interactions between the PUT and the environment is a significant challenge. If the symbolic execution system works on the user mode process level, dependencies on computations made in, e.g., system calls cannot be captured in branch constraints. KLEE [19] and angr [108] solve this problem by using manually-crafted models of system calls. However, due to the large number of system calls in modern operating systems⁵, providing models for all of them is not feasible. KLEE and angr only provide models for a small number of commonly-used system calls. Also, limitations in the system call models will have a detrimental effect on the symbolic executor’s ability to explore the program state space. For example, Yun et al. [132] noted that an overly simplified model of the `mmap` system call prevented angr from finding a bug that their QSYM tool found. A more rigorous method of handling interactions with the OS is to perform full-system symbolic execution, where all code, including the OS kernel and all user mode processes, can be treated symbolically. The S2E tool by Chipounov et al. implements this approach [25]. The improved accuracy of full-system symbolic execution, however, comes at the cost of significantly higher computational and memory overheads. Therefore, this approach is better suited to testing relatively small code bases that have many interactions with the OS kernel, such as device drivers [71].

Greybox Fuzzing

Due to the limited scalability and high implementation complexity of whitebox methods, greybox fuzzing has recently started to attract significant interest, both in industry and in academia. Greybox fuzzers make use of a simple evolutionary strategy to explore the state space of the PUT and find bugs. Greybox fuzzing works by repeatedly cycling over a corpus of seed files. A file is selected for fuzzing with a certain probability (determined by some scheduling strategy, as discussed in Section 3.2 below). After applying mutational fuzzing to the file, it is used as input to the PUT, while measuring the PUT code coverage. If the mutated input covers new code, it is added to the seed corpus for further fuzzing in the future. This simple evolutionary algorithm allows the fuzzer to “incrementally” solve constraints, and achieve drastically better code coverage than a mutational fuzzer without coverage feedback. While the coverage instrumentation adds a certain amount of overhead, input generation is still dramatically faster than for whitebox fuzzers. Therefore, in many cases the bug finding efficiency of greybox fuzzers meets or exceeds that of whitebox fuzzers [33], [113]. The utility of greybox fuzzers for finding bugs has also been extensively demonstrated in practice. The popular greybox fuzzers AFL [133] and libFuzzer [99] have each found hundreds of bugs, many of which are security-critical, across numerous widely-used open source

⁵For example, the Linux 4.17 kernel for x86-64 has almost 300 different system calls.

programs. However, a crucial factor for the effectiveness of the evolutionary approach employed by greybox fuzzers is the speed at which mutated inputs can be tested. This requires that the PUT does not take long to execute. Typical PUT inputs should also be fairly small, so that mutations can be applied to all (or most) parts of an input without requiring an excessive number of mutation/execution runs. (Recall that mutations can only be applied to a small part of an input at a time, to avoid breaking the structure of the input.) For example, the documentation for AFL recommends that inputs should ideally be less than 1 kB [134]. The AFL user interface will also emit a warning if a PUT takes more than 10 ms on average to execute test cases.

In addition to the requirement for speedy PUT execution, greybox fuzzers also share some limitations with mutational fuzzers. So-called *magic constants* pose a problem for any random input-generation strategy. Magic constants are specific values that are compared against input, such that the input must match the constant exactly for a certain program path to be taken. Randomly guessing the exact value of a long constant (e.g., a 32-bit value) within a reasonable period of time is exceedingly unlikely. Steelix [75] uses static analysis to augment AFL’s branch coverage feedback to also include information on how close the fuzzer is to “guessing” the correct magic constant. The enriched coverage feedback allows AFL to incrementally discover inputs that pass constraints involving magic constants.

Complex nested constraints on different parts of an input are also difficult to solve using random mutation. *Hybrid fuzzing* has been proposed as a way to marry the benefits of greybox and whitebox fuzzing. With this approach, the greybox fuzzer has the primary responsibility for exploring the program state space, while the symbolic execution system mainly serves the purpose of solving constraints that are too complex for the greybox fuzzer to resolve with random exploration. This way, the path-explosion problem associated with whitebox fuzzers is ameliorated, while the greybox fuzzer can capitalize on the whitebox fuzzer’s ability to get past complex input constraints. However, a challenge with hybrid fuzzing is how to decide when to “give up” trying to solve constraints with the greybox fuzzer and use symbolic execution instead. Driller [113] uses AFL in combination with the symbolic execution functionality of angr. When AFL has been unable to discover any inputs that trigger new code coverage for a certain amount of time, symbolic execution is invoked to try to solve branch constraints of branches that AFL has yet failed to cover. QSYM [132] uses a different hybrid fuzzing approach. AFL and a concolic execution system are executed in parallel, sharing and contributing to the same input corpus. Similarly to Driller, the concolic system is used to find inputs that cover branches not yet covered by AFL. QSYM also exploits the synergies between greybox and whitebox fuzzing by employing *optimistic solving* to speculatively solve only the last branch constraint of a program path. This reduces solving time for deep paths, and avoids problems with so-called over-constraining [25]. While not every input that is produced is

guaranteed to actually cover new code when optimistic solving is used, invalid inputs can quickly be discarded by the coverage feedback mechanism that is used by the greybox fuzzer.

An alternative approach to solving difficult constraints in greybox fuzzing is to employ gradient descent. Angora [23] uses dynamic data flow analysis to determine which parts of an input are used in a particular branch constraint, and then attempts to find a satisfying solution using gradient descent.

Other Ways to Improve Fuzzing

Several approaches have been proposed to improve the effectiveness and efficiency of fuzzing that are orthogonal to the fuzzing paradigms described above.

The *scheduling strategy* of a fuzzer (e.g., for deciding how much time to spend on fuzzing a particular seed) can have a significant impact on the rate at which bugs are found. Woo et al. [127] studied several scheduling strategies for maximizing the number of bugs found with blackbox mutational fuzzing within a certain time budget. Böhme et al. [13] enhanced AFL with scheduling strategies based on a Markov chain model of greybox fuzzing. One of the key insights of their work is that inputs that exercise *high-frequency paths*, i.e., paths that are very frequently triggered by mutated inputs, should receive lower priority, since they are likely to represent uninteresting error-handling code. This is based on the intuition that most fuzzed inputs will be detected as invalid by the PUT, while those mutated inputs that are *not* recognized as erroneous are the ones that are likely to reveal bugs, or to improve code coverage.

A popular approach to making fuzzers "smarter" is to use dynamic data flow analysis to guide input generation. BuzzFuzz [43] is a mutational fuzzer⁶ that uses dynamic taint analysis to detect the parts of a seed that influence system APIs, and only applies targeted mutations to those parts of the input. TaintScope [121] uses a similar approach, but it also uses control flow altering to bypass integrity checks based on checksum verification, which would otherwise cause every mutated input to be rejected. After finding a crash-inducing input, TaintScope attempts to "repair" checksum fields in the input using symbolic execution in order to verify that the input triggers the bug also without control flow altering. Haller et al. [50] similarly use DTA to detect the parts of an input that influence array indexes in the PUT, and then perform symbolic execution to try to uncover buffer overflow or underflow vulnerabilities. The greybox fuzzer VUzzer [101] employs DTA to limit mutations to only those bytes of an input that influence branch predicates, and to infer which parts of an input must remain unmodified in order for the

⁶The authors use the term "whitebox fuzzing" to describe their approach, although they do not use symbolic execution. They appear to use the word "whitebox" in a more general sense, meaning any method that does not treat the PUT as a black box.

input to be considered valid by the PUT. Angora [23] also makes use of DTA to determine which bytes of the input influence a particular branch so that its gradient descent algorithm can focus on those bytes.

A limitation of data flow guided fuzzers is that they do not usually consider indirect flows (due to problems with spurious information flows, as discussed in Section 3.1). This means that the analysis may fail to, e.g., identify all branches that depend on a particular part of the input. AFL uses a more general method of heuristically determining which bytes are not likely to be affecting the control-flow of the program, and can therefore be safely excluded from mutation. (For example, pixel data in an image file would typically fall into this category.) When an input is selected for fuzzing for the first time, AFL fuzzes its entire range of bytes, one byte at time. If mutating a particular byte (in isolation) never appears to change the control flow of the PUT, that byte is excluded from further fuzzing. Lemieux and Sen [74] use a similar method to approximately determine which parts of an input cannot be mutated in order for a particular branch instruction to be exercised. This approach is used as part of a strategy to steer the fuzzer towards rarely-exercised branches, in order to improve code coverage of the PUT.

An alternative to state-space exploration methods like grey and whitebox fuzzing is automatic learning of input formats. A learning algorithm is used to infer a model of the syntactic and semantic rules of the input format, based on a set of example inputs. The inferred model can then be used to generate inputs, similarly to generational fuzzing. Hörschele and Zeller [55] apply DTA to detect how different parts of an input are used by a program, and then use this to infer a context-free grammar that describes the input format. Wang et al. [119] noted that input generators based on a simple context-free grammars are often able to pass the *syntactic* checks for highly-structured input formats, but still typically fail to pass *semantic* checking. They propose *probabilistic context-free grammars* as a formalism for learning both semantic and syntactic rules from a set of input samples and an existing context-free grammar. They use inferred models to generate seed inputs for AFL, and show that their approach increases code coverage significantly over using the training samples directly as seeds. Some works also use standard machine-learning methods to learn input formats. Godefroid et al. [47] train a recurrent neural network to model PDF files. The model is then used as part of a blackbox mutational fuzzer, thereby increasing the likelihood that introduced mutations are "legal" according to the PDF standard. Böttinger et al. [15] propose training a blackbox fuzzer based on coverage feedback, using deep reinforcement learning.

3.3 Trace Alignment and Binary Code Matching

In this section, we discuss the two related problems of *binary code matching* and *instruction trace alignment* (ITA). The goal of the former is to locate equivalent (or similar) pieces of binary code in two executables, whereas the goal of the latter is to align the execution histories of two binary programs. Binary code matching thus allows comparing the syntactic or semantic similarity between the *code* of two binaries, while ITA allows comparing the *behavior* of binary programs. While the focus of this thesis is dynamic analysis to facilitate instruction trace alignment, we also briefly survey methods for binary code matching.

Binary Code Matching

Matching binary code requires some kind of similarity metric to allow comparison of two code units (e.g., functions or basic blocks). Such similarity metrics can be based on either syntactic or semantic features of the code, depending on the purpose of the code comparison.

In *binary diffing* the goal is to find small differences between otherwise syntactically similar binaries, akin to using the Unix `diff` tool on a source code file. A common use-case of binary diffing is analyzing differences between two versions of the same program, e.g., for finding out exactly what changes a software patch for a closed-source program has introduced. It should be noted, however, that computing a binary diff is significantly harder than doing the same for a source code file. Recompiling a program after a small change may yield a new binary with, for example, different code ordering, different register allocations, and different memory offsets throughout large sections of its code. Therefore, the new binary may superficially look completely different from the original. The BinDiff tool [141] is considered to be the industry standard tool for binary diffing. BinDiff relies on IDA Pro for disassembly and CFG recovery, and uses a heuristic graph-matching algorithm to first align the static call graphs of both binaries using structural information about functions (e.g., the number of basic blocks and function calls) for node matching. The intraprocedural CFGs of matched functions are then also heuristically aligned to discover code differences. Several improvements to the BinDiff approach have been proposed. For example, Bourquin et al. use a more powerful graph-matching algorithm for binary diffing [17], and Gao et al. propose the use of symbolic execution to semantically compare basic blocks, in addition to also using a more accurate graph isomorphism algorithm [44]. Despite these improvements, a fundamental limitation of binary-matching methods based on graph isomorphism is that they require the compared binaries to be highly syntactically similar. Using a different compiler or optimization level may significantly change the structure of the CFG, which will in turn often dramatically reduce the matching accuracy of tools like BinDiff [35].

Another use case for matching executable code is binary code search. Given a piece of binary code, the objective here is to find semantically identical pieces of code in a large set of binary files. For example, given (the machine code of) a version of a library function with a known vulnerability, one might be interested in scanning a large collection of firmware images to determine whether or not the vulnerable version of the code is present in any of the images. In contrast to binary diffing, there is often no requirement with regard to syntactic similarity when performing a binary code search, as the corpus of searched binaries may have been compiled by different compilers or for different CPU architectures. Therefore, methods for binary code search often emphasize semantic features for matching. Luo et al. used symbolic execution to detect semantically identical pieces of code in obfuscated binaries [79]. However, due to the high computational cost of solving SMT queries, methods based on symbolic execution tend to have poor scalability. Chandramohan et al. proposed ameliorating this problem by pre-filtering potential matches using similarity scores based on API-calls and the types of instructions used [22]. David et al. [30] instead address the scalability problems with using theorem provers by breaking functions down into many small units of code called *strands*, and using a theorem prover to quantify the similarity between individual strands. Statistical methods are then used to aggregate strand similarity into a similarity metric for entire functions. Instead of using theorem provers to test semantic similarity, some works use the observed results of runtime computations to build semantic signatures for matching. Egele et al. [35] use Pin to collect the set of all runtime-observed results of computations for every function in a binary. The per-function set of collected runtime-values is then used as a signature to match functions between two binaries. To reach 100% code coverage, some code paths are forced by means of manipulating the execution of branch instructions, and a fixed random memory environment is used to facilitate comparable results across different binaries. In a somewhat similar vein, Pewny et al. [96] create semantic signatures by using code emulation to sample pairs of input/output values for basic blocks. A few works also use structural (i.e., syntactic) features for binary code search. Eschweiler et al. [38] match functions using an approximate graph-isomorphism algorithm, and use pre-filtering based on simple numerical features (e.g., number of instructions, number of basic blocks, etc.) to reduce the time needed for searching. Feng et al., however, found that pre-filtering could lead to unreliable results [39]. They instead proposed representing CFGs as numeric feature vectors to address scalability.

Finally, Zhang and Gupta [138] proposed a method for comparing semantically identical binaries in the presence of syntactic differences due to different levels of code optimization. Their method was later extended by Nagarajan et al. to also handle certain forms of code obfuscation [88]. A mapping between the instructions in two binaries is computed by executing the binaries on the same input, and using several heuristics to match instructions. Match

candidates for each instruction are selected based on how similar their sets of runtime-observed values are (similar to Egele et al. above). Preliminary matches are then further pruned by using a heuristic alignment of the dynamic data dependence graphs of the two binaries. The work by Nagarajan et al. extends this approach to deal with obfuscated code, where the original control flow structure of the program may have been significantly altered. To aid in comparing binaries in the presence of obfuscation, they also produce mappings between nodes of the CFGs in the two binaries. The dynamic call graphs of the respective program executions are aligned heuristically, and the original method by Zhang and Gupta is used for intraprocedural matching of instructions. The instruction mappings are then used to match control flow paths in the CFGs of the two binaries. In contrast to the other binary-matching methods mentioned here, these two methods compare two concrete executions to create instruction mappings, and can therefore only match instructions that are exercised by the given input. (Note, however, that in contrast to ITA, their method still matches *static instructions* rather than instruction execution instances.) Another limitation is that both binaries are required to be semantically identical, as a single discrepancy in the dynamic dependence graphs can lead to propagating instruction mismatches.

Instruction Trace Alignment

While basic notions of syntactic and semantic similarity from binary code matching can also be used in ITA, the two problems offer distinctly different challenges. In binary code matching the units of comparison are typically large (e.g., entire functions), while in ITA the basic unit of comparison is a single instruction. This means that much more informative semantic or syntactic signatures can be extracted in binary code matching, facilitating unique identification of corresponding units of code in matched binaries. On the other hand, since an instruction trace is a runtime artifact, runtime information can be used to aid matching in ITA. In particular, the temporal order of instruction execution is an important additional source of information. However, the greatest challenge in ITA is scalability. Traditional sequence alignment methods, such as the well-known longest common subsequence algorithm, use dynamic programming and therefore have quadratic time complexity. This makes them ill-suited for aligning instruction traces, which often have many millions of elements. For this reason, the existing body of work in ITA is limited. Also, due to the scalability problems of applying general-purpose sequence alignment algorithms for ITA, most existing works use highly application-specific "one-off" solutions.

Kang et al. [62] use trace alignment to diagnose how environment-aware malware⁷ determines that it is being run in an analysis environment. They

⁷Malware that tries to discover if it is being executed in an analysis environment, so that it can refrain from revealing its malicious behavior.

collect traces from two environments, one where the malware exhibits malicious behavior, and one where it does not. After heuristically identifying the divergence point, i.e., the point where the behavior of the malware starts to differ significantly between the two executions, they align the traces from the start of execution up to this point of divergence. This allows identification of the point at which the malware fingerprints its execution environment, since it will result in a small misalignment between the traces. Since it can be expected that the parts of the traces that are subject to alignment will have few differences, a special-purpose alignment algorithm [87] with time complexity $\mathcal{O}(nd)$ can be used, where d is the number of differences and n is the number of instructions. Johnson et al. [60] also consider the problem of aligning two executions of the same binary with slightly different inputs, and show applications in diagnosing the root cause of security vulnerabilities and understanding environment-aware malware. Their method uses *execution indexing* [128] as the basis for their alignment algorithm. Execution indexing uniquely identifies an execution point based on its position within the hierarchical control-dependence structure of the program. It works by dynamically maintaining a stack containing the branches that the current instruction is control-dependent upon. The contents of this control-dependence stack at any point during execution constitutes that point's *execution index*, and can be used to identify the same execution point between two runs of the program. Due to the hierarchical nature of (structured) programs, if two program traces become misaligned at some point, they are guaranteed at some later point to re-align at an earlier execution index (i.e., an index higher up in the hierarchy). Therefore, an efficient linear-time algorithm can be used to align execution indexes of two traces. However, since tracking dynamic control dependencies requires static control flow information, inaccuracies in CFG recovery may adversely affect the correctness of the alignment. This could especially be a problem when analyzing malware, which is frequently obfuscated. Also, the alignment method is only applicable to traces from the exact same binary code, since execution indexing requires identical control-structure in compared executions. Weeratunge et al. [122] use an approach similar to Johnson et al. for aligning traces as part of their method for analyzing concurrency bugs. Their method, however, works on source code and not binary code.

Some works also use sequence alignment techniques to align more coarse grained execution traces. Kirat and Vigna [65] describe a method to create evasion signatures of environment-aware malware, where they use the Smith-Waterman [110] dynamic-programming algorithm to align traces of system calls. System call alignment is useful for comparing the high-level behavior of two programs, since it captures similarities in how programs interact with the environment. Because system call traces are much shorter than full instruction traces, they can also be aligned within reasonable time using classical alignment methods. Ming et al. adopt Kirat and Vigna's approach to align-

ing system call traces in their system for performing semantic comparison of malware behavior [84]. After aligning system calls, dynamic slicing and symbolic execution are used to assess which aligned system call pairs have semantically-equivalent input parameters. Based on this, a final similarity score is calculated.

Chapter 4

Summary of Contributions

In this chapter, we give a detailed overview of the contributions presented in the thesis. First, we describe our approaches to improving the scalability of dynamic slicing (*RQ1*) and instruction trace alignment (*RQ2*), respectively. We then present our methods to improve the semantic insight of random fuzzing (*RQ3a*), and to accelerate fuzzing by avoiding random exploration of "uninteresting" parts of inputs (*RQ3b*).

4.1 Scalable Dynamic Slicing

Dynamic slicing methods based on Agrawal and Horgan’s third method (Section 3.1), which first constructs the dynamic dependence graph and then traverses the graph to compute slices, have previously been considered impractical due to high memory overhead. This has lead others to develop new methods that either try to compress the graph [137], or compute slices from more compact execution traces [136]. These alternative methods are not ideal, however, as the first approach requires a lengthy preprocessing step to compact the graph, while the second approach suffers from slow slice-computation. Moreover, while the graph-compression method by Zhang and Gupta [137] achieves an impressive 94% size reduction on average, their method requires access to source code. In their dynamic slicing system for debugging, Nagarajan et al. [89] used a subset of Zhang and Gupta’s compaction methods that are also applicable on the binary code level. They found that this only resulted in a 70% size reduction. (In other words, five times as much space was required for the graph.) Therefore, they had to apply application-specific optimizations, in addition to limiting slice computations to a small window of the execution history.

The reason that the graph-based approach is considered impractical is that graphs for realistic executions frequently become too large to fit in RAM. Therefore, methods that compute slices from traces have been considered more practical for two main reasons. First, traces are both inherently more

compact, and can also be traversed directly in a compressed format [120]. Second, even in cases where the entire execution history does not fit in RAM, traces are ideally suited to be accessed from secondary storage, since they are completely “flat”, and only a single linear sweep over the trace is required for computing a slice. Overhead from random-access to the secondary storage is therefore naturally avoided with the trace-based method. In contrast, storing and traversing graphs on secondary storage without excessive random-access overhead is a challenging problem. We show, however, that by carefully designing the graph representation to exploit *temporal locality* in programs, it is possible to utilize secondary storage for the dependence graph, without significant slowdowns from random access to the storage media.

In this section, we first provide an overview of the method for backwards slicing described in Paper I. We then briefly discuss the dynamic slicing system that is used in Paper IV. Although both systems are based on a similar design philosophy, the aim of the work in Paper I was to maximize the efficiency of backwards dynamic slicing when utilizing secondary storage, while the second system has been tailored more towards flexibility, and instead supports forward dynamic slicing.

In the implementation of both dynamic slicing systems we have focused on data dependencies only, i.e., our dependence graphs only consist of data dependencies. The reason is twofold. First, recording dynamic control dependencies requires an accurate control flow graph for the program. Since one of our explicit goals has been to perform dynamic slicing at the binary code level, the accuracy of dynamic control dependency analysis will be limited by the fundamental challenges in CFG recovery from binaries. Our second and primary reason for focusing on data dependencies is the problem with spurious dependencies when performing information flow analysis on binary code (Section 3.1). Our decision to not consider control dependencies was reinforced by the observation that, for many applications of dynamic information flow analysis, data dependencies are considered sufficient. For example, most DTA implementations only consider data dependencies. Also, in a study by Zhang et al. [135] on the effectiveness of dynamic slicing for diagnosing bugs, data dependency slicing was found to be sufficient for locating the root cause of about 2/3 of all bugs considered in the study, including all of the memory corruption bugs. It should be noted, however, that our decision not to consider control dependencies is not due to some fundamental limitation of our approach to dynamic slicing. For both of the systems presented in this chapter, the dependence graph representations and the associated methods for construction and traversal of the graphs could easily be extended to incorporate dynamic control-dependence information. Since control dependencies only contribute to about 10% of the size of a dynamic dependence graph [137], we do not expect that adding control dependence edges would require re-design of our methods in any way, nor would it significantly invalidate the empirical performance figures presented in Paper I.

Backward Slicing

The benefit of using traces to represent execution histories for dynamic slicing is that a trace can easily be split up into "chunks", which can be cached in RAM. This allows amelioration of random-access latencies, both during the recording of an execution history, and during the subsequent traversal of the recording to compute slices. The problem addressed in Paper I is to design a physical storage format for execution histories that allows efficient caching in RAM while still retaining the graph representation of data dependencies. Here, we give an overview of our proposed solution to this problem.

A key design decision for our dynamic slicing method was to maintain separate execution histories for each basic block. One entry in a per-basic-block execution history contains references to the defining instruction of each input operand for each instruction in the basic block. This design naturally allows exploiting *temporal code locality*. Only a small subset of all of a program's basic blocks, referred to as the *code working set*, will be executed during a certain (small) time window (for example, the body of a loop with many iterations). With our design, only the execution histories of the current working set need to be maintained in RAM at any given time. We also exploit *temporal locality of reference* of data, i.e., the observation that only a small subset of all data in memory, referred to as the *data working set*, tend to be referenced during a certain time window. This means that data dependencies corresponding to computations on the current data working set will be *physically close* in our per-basic-block execution histories. The execution histories can therefore be split into chunks, so that only a few recently used chunks for each basic block in the code working set need to be cached in RAM during traversal of the graph.

Another design goal of our work was to avoid the expensive preprocessing step of Zhang and Gupta's method. To achieve this, the graph is created in an online fashion during the execution of the target program. Dynamic dependencies are recorded into chunk buffers, which are committed to disk when they become full. We present an algorithm for dynamically estimating the code working set, so that chunk buffers only need to be maintained for the most recently executed basic blocks. Given a certain memory budget, the size of chunk buffers, and the number of basic blocks for which to maintain a chunk buffer in memory, are dynamically adjusted to make optimal use of RAM. In designing this algorithm, we also needed to consider that the execution frequencies of basic blocks follow a heavy-tail distribution, where a few basic blocks will be responsible for the vast majority of the execution time, while a large number of basic blocks execute quite rarely. As we show in the paper, fixed-size chunks therefore result in poor utilization of RAM and drastically reduced performance. The size reserved for a chunk buffer during graph construction must instead be dynamically adjusted based on the current execution frequency of the corresponding basic block. Using results

from network communication scheduling [116], we show that the random-access overhead of committing chunks to disk is minimized if the chunk size is proportional to the square root of the basic block execution rate. This property is achieved if the size reserved for chunks of a particular basic block is incremented with a constant each time the current chunk must be committed to disk.

The graph-recording component of our system was implemented with Pin. Code to manage chunk buffers and record data dependency information is added to each dynamic basic block using Pin’s instrumentation facilities. A shadow state is used to hold a reference to the last defining instruction of each register and memory location, so that this information can be recorded into the graph when the corresponding real location is dereferenced. During the instrumentation phase, we also perform static analysis on dynamic basic blocks to implement two of the compression methods from Zhang and Gupta’s work [137]. These work by identifying static data dependencies within the basic block that do not need to be explicitly recorded, resulting in an average graph size reduction of 30% in our experiments. Since the two methods only require intra-basic block analysis, they are also applicable to binary code and do not necessitate any expensive preprocessing of the execution history before slice computation.

Our implementation was empirically evaluated using five popular Linux programs. For each program, we recorded dependence graphs for executions consisting of several billion instructions, resulting in graphs 21–72 GB in size. The mean overhead for online graph recording was 100x across the five programs. To evaluate the slicing performance, we computed 1000 slices at random points during the execution history. Using 2 GB of RAM for caching, the cache miss ratio¹ was between 0.3% and 0.002%, showing the effectiveness of our design’s ability to exploit temporal locality. This allowed graph traversal rates of 5–9 million dependence edges per second.

Forward Slicing

In contrast to the system described above, the dynamic slicer used in Paper IV is designed to support forward slicing. To compute forward slices, we need to record forward dependence edges in the dynamic dependence graph. That is, for each instruction we need to create edges to all instructions that *later* used a variable defined by that instruction. Since the number of forward edges of a node is not known a priori, we cannot easily reuse our graph representation for backwards slicing. Instead, we adopt a simplified design in which each instruction execution results in one physical node being created, and where nodes are stored linearly to file as they are created. To support an arbitrary number of forward edges, if a node has more than one outgoing edge, we create

¹The proportion of dependence edge lookups that required fetching a chunk from disk.

a linked list of edges. Nodes of the linked list are stored in-line with the nodes of the graph on file. To handle graphs that are too large to fit in RAM, we use memory-mapped I/O to map the entire graph file into virtual memory, during both construction and traversal. In cases where the graph is too large to fit in RAM, we rely on the operating system’s page handling mechanism to keep only recently-used parts of the graph in memory. Note that, while this simple design is not as efficient as our backwards slicer, we still rely on the same fundamental observation that temporal locality of reference can be used to avoid excessive random access to secondary storage. Since instructions referencing the current data working set will execute close to each other in time, they will also be clustered physically close to each other on file. This means that with a depth-first graph traversal strategy, only a small number of memory pages from the backing graph file need to be mapped into RAM at any given time.

Just like the backwards slicing system, we also implemented the binary analysis part of our forward slicer using Pin. A limitation of our design is that, in order to support graphs larger than 4GB, the graph construction and traversal need to run on a 64-bit operating system. To improve the flexibility of our system, and also to allow for slicing on 32-bit systems, we use an implementation similar to offline taint analysis [124]. Instead of creating the graph online during execution of the target program, we record a trace containing all runtime information necessary to resolve dynamic data dependencies. The trace is then used to efficiently build the graph, potentially on another system.

4.2 Scalable Instruction Trace Alignment

Instruction trace alignment allows fine-grained comparison of two program executions, which is important in several applications. For example, malware authors often distribute slightly modified (i.e., obfuscated) variants of the same malicious binary in an effort to circumvent signature-based antivirus products. Similarly, instruction trace alignment could help to identify program plagiarism or aid in general program-comprehension tasks. In contrast to the highly application-specific methods from earlier works, which have good accuracy for specific applications, but are not applicable in the general case, the goal of our work in Paper II has been to devise a *general* trace-alignment method. Due to either *syntactic differences*, that stem from, e.g., different optimization levels or use of different compilers, or *semantic differences* that occur when, for example, comparing two versions of the same program, a perfect alignment cannot be expected to exist in the general case. Therefore, we have striven to devise a *best effort* alignment method, which can produce meaningful alignments even in the presence of significant syntactic differences, or minor semantic differences. To the best of our knowledge, we are the first ones to have addressed this general trace-alignment problem.

Figure 4.1: Schematic example of the FastDTW alignment algorithm. Red cells represent the current alignment. White cells represent the part of the alignment matrix considered during one round, while grey cells represent parts of the matrix that are ignored.

Classical dynamic-programming sequence alignment methods work by finding a minimum-cost path through an alignment matrix, where the cost at each cell is determined by the level of dissimilarity between the corresponding elements of the two aligned sequences. Since such methods have quadratic time complexity, they are not directly applicable for aligning instruction traces. We instead adopt the FastDTW [104] alignment method, which is a heuristic linear-time variant of the quadratic-time *dynamic time warping* (DTW) algorithm. DTW is frequently used in, e.g., speech recognition to align time series representing analog signals [86]. Figure 4.1 shows a simple example of the FastDTW algorithm in action. FastDTW works by aggregating adjacent elements of each sequence into larger units to create a low-resolution version of the alignment problem. After computing a low-resolution alignment, the resolution is increased, and a new alignment is performed only for a narrow band around the initial alignment. In the context of aligning analog time series, aggregating several elements of a sequence is done simply by computing an average over the elements. In our context, however, sequences consist of machine code instructions. Therefore, a key design problem was to identify a feature that (i) captures the *semantics* of computations, rather than syntactic properties, and (ii) lends itself well to aggregation over a sequence of instructions. We choose, similar to some previous works on dynamic code-matching [138, 35], the set of *runtime observed values* for each instructions (i.e. values of input and output operands) as our code-matching feature. Runtime-observed values fulfill requirement (i), since two semantically similar programs must compute a similar set of intermediate results in roughly the same order to arrive at the final output. Requirement (ii) can also be fulfilled by comparing the frequency-distribution of observed values for a range of instructions. To improve the matching accuracy, we

also employ the tf-idf technique from information retrieval to weight value frequencies according to the value’s level of uniqueness.

For the empirical evaluation of our method we implemented a tracing component using Pin. We also use a heuristic technique to filter out values that are likely to be addresses (i.e., pointers), which are generally not comparable across executions. The trace alignment method was evaluated on 11 Linux programs. We first compiled the programs using different compilers and compiler settings to produce a set of syntactically different binaries for each program. For each of the programs, we then recorded traces from executing the different binaries on the same input, and evaluated how accurately traces from different binaries could be aligned. The alignment accuracy was tested across different compilers, different optimization settings, and between 32-bit and 64-bit x86 binaries. We found that for the vast majority of instructions in a trace, the produced alignments were off by a few instructions at most. We also evaluated the method on obfuscated code, and found that for many programs it could still produce meaningful alignments, albeit with lower accuracy. The performance of the implementation was also evaluated. For short traces (a few hundred thousand up to a few million instructions), traces could be aligned in times ranging from a few seconds to a few minutes. For longer traces of up to about 100 million instructions, alignment times were about 30–60 minutes, showing the linear scaling of our approach. In addition to the quantitative evaluation of our method’s accuracy and performance, we also show a practical use case of applying trace-alignment to study semantic differences between two versions of the same program compiled with different compilers and optimization levels.

4.3 Improving Fuzzing using Dynamic Slicing

In this section, we give an overview of our efforts to improve fuzzing. First, we present a system called *MutaGen* (Paper III), which implements a novel method of improving code coverage by turning a generating program into a generational fuzzer of sorts. We then describe our approach to accelerating greybox fuzzing of slow PUTs, or PUTs that require large inputs (Paper IV).

Dynamic slicing is an integral part of both of our contributions to fuzzing. Apart from allowing fuzzing of closed-source programs, performing dynamic slicing using dynamic binary instrumentation also significantly simplifies deployment of our fuzzing systems compared to a solution that applies instrumentation to the source code or during compilation. These latter two approaches would require not only the PUT itself, but also all shared libraries used by the PUT, to be recompiled with instrumentation to track data dependencies. It is not uncommon for programs to link to tens, or sometimes even hundreds, of shared libraries. Note that, even if we are not interested in computations that take place inside a library, it is also crucial to record

data flow inside the libraries, so as not to break the tracking of transitive data-flows in the PUT. If data flows from the PUT into a shared library, for example when the PUT calls a library function, we must also instrument all computations inside the library to record data dependencies, otherwise we would not be able to establish the dependence between the data flowing out of the library (e.g., a return value), and the input to the library function.

MutaGen

The aim of our MutaGen system was to combine the low human effort of mutational slicing with the superior code coverage of generational fuzzing. In generational fuzzing, code coverage is improved by increasing the semantic insight of the fuzzer by providing a manually crafted specification of the PUT’s input format. However, crafting the input specification is a laborious and error-prone task. The key insight behind our method is that for most input formats — even proprietary or undocumented ones — there is a *generating program*, which can create correctly formatted inputs. (Although the generating program may only be distributed in binary form for proprietary formats.) Such a generating program *implicitly* encodes the input format in its output-generation code. Our method is inspired by mutation testing [59], where mutations are introduced into a code base in a systematic fashion in order to evaluate the completeness of an accompanying test suite. We instead apply systematic code mutation to a generating program, so that its outputs become *slightly* corrupt. The outputs of the mutated generating program is then used as inputs to the PUT. The hypothesis of our work has been that, in contrast to “blindly” mutating an input directly, our mutations take place at a higher level of abstraction, increasing the likelihood that generated inputs pass syntactic and semantic checks of the PUT. In that sense, our method turns an existing generating program into a generational fuzzer.

Figure 4.2 illustrates the difference between traditional mutational fuzzing and MutaGen. While a mutational fuzzer blindly introduces “noise” into a seed file, MutaGen conceptually works by injecting “bugs” into the output generation part of a generating program. In the example depicted in the figure, MutaGen is used to generate inputs for testing PDF parsers². A PostScript-to-PDF converter is used as the generating program. MutaGen first applies a mutation to the PostScript-to-PDF converter and then executes it with a valid PostScript file as the input. The resulting output can then be used as a test input for PDF readers. By repeating this process for different mutation sites in the generating program, a large corpus of test inputs can be generated.

MutaGen is implemented to introduce mutations directly to x86 binaries using Valgrind. Applying mutations directly at the machine code level avoids difficulties with programs that are made up of many different components,

²Note, however, that our method is applicable to all programs that take files as input, not just PDF readers.

Figure 4.2: Conceptual difference between mutational fuzzing (a) and MutaGen (b), for the example case of generating test inputs for PDF readers.

and also allows MutaGen to support closed-source generating programs. Using Valgrind allows us to avoid the intricacies of the x86 instruction set, and apply mutations on Valgrind’s simplified IR. We use several mutation operators from mutation testing, such as switching addition and subtraction. In contrast to mutation testing we apply mutations to *computations* rather than branch predicates, since our goal is to mutate the computed output of the generating program rather than drastically changing its internal logic. Therefore, we also use several arithmetic mutation operators, for example adding or subtracting a constant from different instruction operands. Since applying mutations to every executed instruction of the generating program would be very time consuming, we also utilize our backwards dynamic slicer from Paper I to limit the set of instructions that are viable for mutation. We instrument system calls for writing output to a file, and treat every byte of the generating program’s output as a combined slicing criterion for backwards slicing. This means that the slice will contain every instruction that is directly involved in computing the program’s output. (Or more precisely, every instruction that at least one byte of output has a transitive data dependency on.) We found

in our experiments that this reduced the number of instructions to mutate by 90% or more.

We evaluated MutaGen on several Linux programs, using 8 programs as fuzzing targets, 6 generating programs, and different inputs and configurations for the generating programs. Several thousand test inputs were obtained from each generator configuration, after removing non-unique files from each set. MutaGen found a total of 16 crashes, 8 of which represented unique bugs. Several of the bugs were memory errors with potential security implications. We also compared MutaGen to two popular mutational fuzzers, which were executed for 24 hours using outputs of the non-mutated generating programs as seeds. Neither of them found any bugs in the 8 programs used in our evaluation. The code coverage achieved by MutaGen was also about one order of magnitude greater than for the mutational fuzzers.

Focused Fuzzing

A prerequisite for the efficiency of the evolutionary approach used by greybox fuzzers is that the PUT must execute reasonably fast. Also, inputs should be fairly small to allow fast exploration of the input space. For "heavyweight" programs, or programs for which even "minimal" inputs are fairly large, greybox fuzzing is markedly less effective. Document viewers and media parsers are examples of programs which are often challenging to test efficiently using greybox fuzzing, for the above reasons. The aim of our work has been to boost the efficiency of greybox fuzzing for such cases by concentrating the fuzzing effort on "interesting" parts of inputs. Specifically, we have focused on accelerating the so-called *deterministic stage* of the state-of-the-art greybox fuzzer AFL. In order to avoid the manually-specified fuzzing ratios of mutational fuzzers, AFL exhaustively applies a regimen of mutations at every byte offset of an input the first time that input is selected for fuzzing. (Subsequent fuzzing of the input will apply a random number of mutations at random offsets, like in classical mutational fuzzing.) This eliminates the risk of missing a bug because an important part of the input never happened to be subjected to mutation. However, the deterministic stage is very costly for "heavyweight" PUTs or large inputs, since AFL would need to perform several hundred mutation/execution runs per byte of input. As a fall back for cases in which the deterministic stage is prohibitively expensive, AFL provides an option to skip it altogether and instead to always apply mutations more like a classical mutational fuzzer. This allows AFL to more quickly cover new parts of the PUT's state space, at the expense of an increased risk of missing bugs.

The foundation of our method is to extend the notion of code coverage to *individual bytes of input*. Greybox fuzzers like AFL already consider code coverage at the granularity of entire inputs — each input that exercises new code is considered interesting and is saved for further exploration. We extend this idea to mark individual bytes of an input as more interesting to

explore by using targeted mutation. The first time a new input is selected for fuzzing, we compute individual forward dynamic slices for each byte of input (i.e., for each input we compute the set of instructions that has a transitive data dependency on that byte). If a slice contains an instruction not previously seen in slices for any previous input, the corresponding byte is considered "interesting". During the subsequent deterministic fuzzing stage, only the "interesting" bytes are subjected to mutation. We dubbed this approach *focused fuzzing*. The system described in Section 4.1 was used to compute the forward dynamic slices. Since the purpose of our work is to accelerate the fuzzing process, it is important that the overhead of dynamic slicing is not so great that it outweighs the benefits of our per-byte coverage feedback. While computing dynamic slices for all bytes in a large input is quite expensive, we observed that in many cases, some nodes in the dynamic dependence graph had a transitive data dependency on a large number of input bytes. Rather than re-traversing these nodes for every slice computation, we utilize a slice caching mechanism. When the same node has been visited in more than a set number of slice computations, we compute the dynamic slice of that node and store it in a cache. If the same node is encountered in future slice computations, we refrain from traversing down that part of the graph, and instead simply merge the cached slice into the current slice. This optimization reduces the overhead of slicing everywhere from a few times up to several orders of magnitude, depending on PUT and input. As a further optimization, we only generated nodes in the dynamic data dependence graphs for instructions that were tainted by input, which significantly reduced the size of graphs. As a result of these optimizations, the cost of computing slices in our experiments was quite reasonable, with the average time to compute all slices for an input being on the order of 10 seconds.

We evaluated our focused fuzzing implementation on 4 Linux programs (two document parsers, one media decoder and one file compression tool), comparing our approach with both the default mode of AFL (which performs the deterministic stage) and the fallback mode. To allow a fair comparison with the default mode, which requires more time to reach reasonable code coverage when using "heavyweight" programs, we ran each fuzzer mode for one week per program. We found that the rate at which focused fuzzing increases code coverage is comparable to AFL's fallback mode, but that focused fuzzing found significantly more crashing inputs and unique bugs. The default mode of AFL had a significantly slower rate of code coverage increase and found far fewer crashes and bugs. It should be noted, however, that focused fuzzing is a heuristic to speed up fuzzing, and is therefore not guaranteed to discover all bugs that are discoverable when using the full deterministic stage. This is evident from the fact that one of the bugs found in our experiment was only discovered by the default AFL configuration. However, given the much higher overall bug and crash count, focused fuzzing appears to strike a good balance

between the speed of the fallback mode and the thoroughness of the default mode.

Chapter 5

Conclusions and Future Work

Being able to perform analysis directly on binary code has many important applications, for example auditing closed-source software for security problems and defending against malware. In many cases, deployment of program analysis tools is also easier when directly analyzing binaries, since one need not consider the idiosyncrasies of different programming languages, build environments, or third-party components. Static and dynamic binary analysis methods complement each other in terms of strengths and weaknesses. Static methods are often more scalable and easier to deploy, and may be the only choice if code cannot be instrumented easily for runtime analysis, as is sometimes the case with, e.g., embedded systems. However, due to the inherent challenges of statically recovering high-level semantics from binary code, fine-grained program analysis, such as information flow tracking or instruction-level code matching, require the use of dynamic analysis. While the main challenge of static binary code analysis is devising effective heuristics for recovering high-level information, *scalability* is instead often the main challenge when doing fine-grained dynamic analysis. In this thesis, we have proposed ways of improving scalability for two important fine-grained dynamic analysis methods, *dynamic slicing* and *instruction trace alignment*, which have not previously seen much use in practice, in part due to their poor scalability. We have also applied our work on scalable dynamic slicing in two novel approaches to improve the effectiveness and efficiency of fuzzing. In our first work on fuzzing, we demonstrated how to improve the semantic insight of blackbox fuzzing by automatically turning an existing program that can generate valid inputs in the desired format into a generational fuzzer. Our second work proposed a method for boosting the efficiency of fuzzing by focusing the random exploration effort on previously unexplored parts of the target program’s input space. In this chapter, we first discuss the conclusions of our work. We then outline some interesting directions for future work.

5.1 Conclusions

Zhang et al. [136] put forward the notion that dynamic slicing approaches based on building the dynamic dependence graph first are impractical, because graphs are often too large to fit in RAM. This statement has also been reiterated in other prominent work on dynamic slicing [120]. The main contribution of our work on dynamic slicing presented in this thesis is to show that graph-based methods can indeed be practical, even when graphs are too large to store in RAM. By designing data structures and algorithms to exploit temporal locality in programs, we can achieve efficient utilization of secondary storage for dynamic slicing. The graph-based approach is especially efficient when many dynamic slices need to be computed from the same execution, as was the case in both of our fuzzing-related applications. In addition to the empirical evaluation of our backwards slicing method presented in Paper I, the demonstration of its utility as part of the MutaGen system presented in Paper III further indicates the practicality of our approach. The forward dynamic slicer used in Paper IV represents a simpler design, albeit one that is still based on the same fundamental principle of exploiting temporal locality. In both of our fuzzing applications of dynamic slicing, executions are, for the most part, short enough that dependence graphs can fit in main memory. However, the ability to gracefully scale to cases where graphs become very large is important in order to avoid out-of-memory failures during fuzzing. Since the explicit objective of fuzzing is to try to trigger undefined or erroneous behavior in the PUT, a fuzzing system must be able to handle PUT executions with abnormally high runtime or memory use, which would typically result in very large dynamic dependence graphs. This is in line with the general goal of avoiding as many failure modes as possible when designing fuzzers, as mentioned in Section 1.5.

In this thesis, we have also presented what is, to the best of our knowledge, the first general method of aligning program traces on the instruction level. In contrast to previous works, which sacrifice generality to be able to exploit application-specific properties of traces in order to improve alignment accuracy, we have striven to develop a maximally general *best-effort* trace alignment method. We have shown that our method can produce meaningful alignments even in the presence of significant syntactic differences. By *meaningful*, we mean that our method gracefully handles non-alignable or re-ordered regions without degenerating to meaningless global alignments. We measured this property by considering how far individual instruction alignments were from the ideal alignment, and found that most alignments were indeed close to ideal. We hope that our method could serve as a foundation for future work on trace alignment, and that our general design could be extended or adapted to different usage scenarios. We discuss a few such potential future directions of work in the next section.

In addition to the first two papers in this thesis, which were focused on method development to address fundamental scalability challenges in dynamic analysis, we have also striven to use the methods we have developed in more applied research. Specifically, we have proposed ways of improving fuzzing using dynamic slicing. In Paper III, we tested our hypothesis that mutations applied to a generating program’s code would result in mutated test inputs that more closely adhered to the syntactic and semantic rules of the program under test. Our experimental evaluation strongly indicated the validity of this hypothesis, as our MutaGen system achieved about one order of magnitude better code coverage than two mutational fuzzers. Also, neither of the mutational fuzzers found any of the 8 bugs discovered by MutaGen.

Paper IV describes a different approach to improving fuzzing. Here, we focus on accelerating the rate at which the state-of-the-art fuzzer AFL discovers bugs by using forward dynamic slicing to extend the notion of code coverage to individual bytes of input. This allows us to focus fuzzing on the parts of the input that exercise previously unseen code. We found that our approach achieves a rate of code coverage improvement similar to AFL’s less thorough fallback mode, but finds more bugs and crashes than either the fallback or default mode of AFL. This indicates that our enhanced version of AFL achieves a better trade-off between speed and thoroughness than vanilla AFL.

5.2 Future Work

In this section, we discuss some limitations of our work and propose potential ways to address these limitations. We also outline general directions for future work related to the problems we have considered in this thesis.

Instruction trace alignment. For our trace alignment method, we use runtime-observed values as a proxy for the actual semantics of matched binary code. While our experiments showed that this is a viable approach, it has some limitations. For example, both programs need to be executed using the exact same input, and both must be perfectly deterministic in terms of which concrete computations are performed. Minor semantic changes that somehow change many of the concrete values produced by some intermediate step of an algorithm will also result in unalignable traces. An open problem is to investigate semantic features that are more suitable for matching code under such circumstances. A potential alternative to explore is using data flow relations in a similar vein to Zhang and Gupta [138].

The focus of the work on instruction trace alignment presented here has been to design the first general approach to the problem. We also discussed some applications in, e.g., malware analysis and plagiarism detection. Due to the scalability problems, however, trace alignment has not previously been considered as a viable approach for comparing executions for most applications. Therefore, an important direction for future work is to explore various

applications of our method, or derivatives of it. One interesting potential application is *automatic program repair* [72], where a meta-heuristic search algorithm is used to automatically fix bugs. The objective is to find a new program that is maximally similar to the original in terms of semantics, but does not exhibit the buggy behavior. Trace alignment could potentially be used to compute a similarity metric for performing this comparison.

Due to the generality of our method, it may also be possible to use it for aligning other types of sequences than instruction traces, or to use features other than runtime-observed values for the tf-idf-based matching of elements. In order to use our matching method, the only requirement is that there must be a sufficient "vocabulary" of different values, so that the frequency distribution of those values uniquely identifies a segment of the sequence. For trace alignment problems where the programs that are being compared can be expected to have similar syntax (as is typically the case in, for example, automatic program repair), it may be possible to avoid the overhead of the more "heavyweight" instrumentation required to record runtime-observed values, and to instead just use instruction opcodes for matching.

Fuzzing. There are several interesting directions for future work for our MutaGen approach. In Paper III, we explore the potential for reducing the number of mutation operators to speed up input generation. We found that by using only 3 out of the total set of 17 mutation operators, we achieve 90% of the code coverage of the whole set. Exploring such optimizations, or devising new and more powerful mutation operators, could be an interesting topic for future work.

Wang et al. [119] combined a generational fuzzing approach based on learned format rules with greybox fuzzing, so that inputs generated from syntactic and semantic rules were fed as seed inputs to AFL for further fuzzing. The inputs produced with generational fuzzing offered a diverse coverage of the input space, while random mutations applied by AFL allowed for a more thorough exploration of corner cases. A similar approach could also be used with our work, where MutaGen could be used to create seed inputs for a greybox fuzzer such as AFL.

Since the utility of a fuzzer is, to a large degree, determined by the speed at which it can discover bugs, implementation-level optimization is also of great practical importance. The use of Valgrind facilitated ease of implementation, but is currently a bottleneck in our MutaGen implementation. Performing mutations using, e.g., binary patching could offer better performance. Also, in our focused fuzzing implementation, dynamic slicing incurs a relatively high cost, despite our algorithmic optimizations. One possible optimization is to perform slicing with more coarse-grained slicing criteria, so that "interestingness" is assigned to larger pieces of an input, rather than to individual bytes.

Extended evaluation. Due to the applied nature of the work presented in this thesis, the validity of our contributions largely rests upon the results of

our empirical evaluations. Performing a more thorough evaluation of our work would further strengthen the confidence in our conclusions. A limitation of the evaluation of our trace alignment method is that it is heavily focused on syntactic differences. This was mostly a practical consideration, as reliable ground truth is not readily available when evaluating semantic differences. Nonetheless, studying our method’s resilience to semantic transformations would shed additional light on potential areas of improvement of our work. Another challenge of evaluating the trace alignment method is that, as we are the first to attempt a general solution to the trace alignment problem, no state-of-the-art implementations exist for comparison.

As fuzzing is inherently a stochastic process, it has been pointed out [68] that to get more reliable results, one needs to average the performance of fuzzers over many long-running fuzzing sessions. However, due to the significant computational resources needed, this is not always possible. For example, we expended 16 CPU weeks of computation to evaluate the focused fuzzing approach of Paper IV, running each fuzzer configuration just once. However, performing a more thorough evaluation of our contributions to fuzzing would of course be desirable. Performing a thorough comparison of MutaGen against modern greybox fuzzers, both in terms of performance (coverage, bugs found), but also to determine if the two approaches can find *different kinds* of bugs, would also be of interest.

Bibliography

- [1] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. “Debugging with dynamic slicing and backtracking”. In: *Software: Practice and Experience* 23.6 (1993), pp. 589–616.
- [2] Hiralal Agrawal and Joseph R. Horgan. “Dynamic program slicing”. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 1990, pp. 246–256.
- [3] Dave Aitel. *SPIKE Fuzzer*. <http://www.immunityinc.com/resources/index.html>. 2002.
- [4] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. “An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries.” In: *USENIX Security Symposium*. 2016, pp. 583–600.
- [5] Brad Arkin. *Adobe Reader and Acrobat Security Initiative*. http://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html. 2009.
- [6] Gogul Balakrishnan and Thomas Reps. “WYSINWYX: What You See is Not What You eXecute”. In: *ACM Transactions on Programming Languages and Systems* (2010), 23:1–23:84.
- [7] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. “Strict control dependence and its effect on dynamic information flow analyses”. In: *Proceedings of the 19th international symposium on Software testing and analysis*. 2010, pp. 13–24.
- [8] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. “Byteweight: Learning to recognize functions in binary code”. In: *USENIX Security Symposium*. 2014.
- [9] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. “Scalable, behavior-based malware clustering.” In: *Proceedings of the Network and Distributed System Security Symposium*. 2009, pp. 8–11.

- [10] Jean-Francois Bergeretti and Bernard A Carré. “Information-flow and data-flow analysis of while-programs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.1 (1985), pp. 37–61.
- [11] Andrew R. Bernat and Barton P. Miller. “Anywhere, Any-time Binary Instrumentation”. In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. 2011, pp. 9–16.
- [12] Jonathan Berr. “”WannaCry” ransomware attack losses could reach \$4 billion”. In: *CBS News* (May 16, 2017). URL: <https://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses/>.
- [13] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-based Greybox Fuzzing as Markov Chain”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 1032–1043.
- [14] Erik Bosman, Asia Slowinska, and Herbert Bos. “Minemu: The world’s fastest taint tracker”. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2011, pp. 1–20.
- [15] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. “Deep reinforcement fuzzing”. In: *Proceedings of the 2018 IEEE Security and Privacy Workshops*. 2018, pp. 116–122.
- [16] Ella Bounimova, Patrice Godefroid, and David Molnar. “Billions and Billions of Constraints: Whitebox Fuzz Testing in Production”. In: *Proceedings of the 2013 International Conference on Software Engineering*. 2013, pp. 122–131.
- [17] Martial Bourquin, Andy King, and Edward Robbins. “BinSlayer: accurate comparison of binary executables”. In: *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM. 2013.
- [18] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. “An infrastructure for adaptive dynamic optimization”. In: *2003 International Symposium on Code Generation and Optimization*. 2003, pp. 265–275.
- [19] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” In: *8th USENIX Symposium on Operating Systems Design and Implementation*. 2008, pp. 209–224.
- [20] Sang Kil Cha, T. Avgerinos, A. Rebert, and D. Brumley. “Unleashing Mayhem on Binary Code”. In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 2012, pp. 380–394.

- [21] Sang Kil Cha, Maverick Woo, and David Brumley. “Program-adaptive mutational fuzzing”. In: *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 725–741.
- [22] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. “BinGo: Cross-architecture cross-OS Binary Search”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 678–689.
- [23] P. Chen and H. Chen. “Angora: Efficient Fuzzing by Principled Search”. In: *2018 IEEE Symposium on Security and Privacy*. 2018, pp. 711–725.
- [24] 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: *2008 IEEE International Conference on Dependable Systems and Networks*. 2008, pp. 177–186.
- [25] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: A platform for in-vivo multi-path analysis of software systems”. In: *ACM Sigplan Notices* 46.3 (2011), pp. 265–278.
- [26] Cristina Cifuentes. “Reverse Compilation Techniques”. PhD thesis. Queensland University of Technology, 1994.
- [27] Cristina Cifuentes and Antoine Fraboulet. “Intraprocedural Static Slicing of Binary Executables”. In: *Proceedings of the International Conference on Software Maintenance*. 1997.
- [28] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Li-dong Zhou, Lintao Zhang, and Paul Barham. “Vigilante: End-to-end Containment of Internet Worms”. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. ACM, 2005, pp. 133–147.
- [29] *Data breach digest - Scenarios from the field*. Verizon, 2016. URL: http://www.verizonenterprise.com/resources/reports/rp_data-breach-digest_xg_en.pdf.
- [30] Yaniv David, Nimrod Partush, and Eran Yahav. “Statistical similarity of binaries”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016, pp. 266–280.
- [31] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [32] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. “Ether: Malware analysis via hardware virtualization extensions”. In: *Proceedings of the 15th ACM conference on Computer and communications security*. 2008, pp. 51–62.

- [33] 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: *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. IEEE. 2016, pp. 110–121.
- [34] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. “A survey on automated dynamic malware-analysis techniques and tools”. In: *ACM computing surveys (CSUR)* 44.2 (2012).
- [35] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. “Blanket execution: Dynamic similarity testing for program binaries and components”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 303–317.
- [36] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005. ISBN: 978-0764574818.
- [37] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *ACM Transactions on Computer Systems* 32.2 (2014), 5:1–5:29.
- [38] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. “DiscovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code”. In: *2016 Network and Distributed System Security Symposium*. 2016.
- [39] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. “Scalable Graph-based Bug Search for Firmware Images”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 480–491.
- [40] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. “The program dependence graph and its use in optimization”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (1987), pp. 319–349.
- [41] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. “SmartDec: approaching C++ decompilation”. In: *2011 18th Working Conference on Reverse Engineering*. 2011, pp. 347–356.
- [42] Vijay Ganesh and David L Dill. “A decision procedure for bit-vectors and arrays”. In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 519–531.
- [43] Vijay Ganesh, Tim Leek, and Martin Rinard. “Taint-based directed whitebox fuzzing”. In: *Proceedings of the 31st International Conference on Software Engineering*. 2009, pp. 474–484.

-
- [44] Debin Gao, Michael K Reiter, and Dawn Song. “Binhunt: Automatically finding semantic differences in binary programs”. In: *International Conference on Information and Communications Security*. Springer. 2008, pp. 238–255.
 - [45] *GNU Binutils*. URL: <https://www.gnu.org/software/binutils/>.
 - [46] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. “Automated Whitebox Fuzz Testing”. In: *Proceedings of the Network and Distributed System Security Symposium*. 2008.
 - [47] Patrice Godefroid, Hila Peleg, and Rishabh Singh. “Learn&fuzz: Machine learning for input fuzzing”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 50–59.
 - [48] Andy Greenberg. “Hackers remotely kill a Jeep on the highway—with me in it”. In: *Wired* (July 21, 2015). URL: <https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/>.
 - [49] Andy Greenberg. “The untold story of NotPetya, the most devastating cyberattack in history”. In: *Wired* (Aug. 22, 2018). URL: <https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/>.
 - [50] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. “Dowsing for overflows: a guided fuzzer to find buffer boundary violations.” In: *USENIX Security Symposium*. 2013, pp. 49–64.
 - [51] Bengt Halvorson. “Software Now To Blame For 15 Percent Of Car Recalls”. In: *Popular Science* (June 2, 2016). URL: <https://www.popsci.com/software-rising-cause-car-recalls>.
 - [52] Sam Hocevar. *zzuf - multi-purpose fuzzer*. <http://caca.zoy.org/wiki/zzuf>. 2010.
 - [53] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with Code Fragments”. In: *Proceedings of the 21st USENIX Security Symposium*. 2012.
 - [54] *Hopper*. URL: <https://www.hopperapp.com/>.
 - [55] Matthias Hörschele and Andreas Zeller. “Mining Input Grammars from Dynamic Taints”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 2016, pp. 720–725.
 - [56] Allen D Householder and Jonathan M Foote. *Probability-based parameter selection for black-box fuzz testing*. Tech. rep. CERT, 2012.
 - [57] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

- [58] Adobe Systems Incorporated. *Adobe Portable Document Format, Version 1.7*. https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdf_reference_archive/pdf_reference_1-7.pdf. 2006.
- [59] Yue Jia and M. Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678.
- [60] Noah M Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. “Differential slicing: Identifying causal execution differences for security applications”. In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 347–362.
- [61] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. “DTA++: Dynamic taint analysis with targeted control-flow propagation.” In: *Proceedings of the Network and Distributed System Security Symposium*. 2011.
- [62] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. “Emulating emulation-resistant malware”. In: *Proceedings of the 1st ACM workshop on Virtual machine security*. ACM. 2009, pp. 11–22.
- [63] Ulf Kargen and Nahid Shahmehri. “InputTracer: A data-flow analysis tool for manual program comprehension of x86 binaries”. In: *IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 2012, pp. 138–143.
- [64] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. “Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems”. In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. ACM, 2012, pp. 121–132.
- [65] Dhilung Kirat and Giovanni Vigna. “MalGene: Automatic Extraction of Malware Analysis Evasion Signature”. In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 769–780.
- [66] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy. “Interprocedural static slicing of binary executables”. In: *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*. 2003, pp. 118–127.
- [67] Ákos Kiss, Judit Jász, and Tibor Gyimóthy. “Using Dynamic Information in the Interprocedural Static Slicing of Binary Executables”. In: *Software Quality Journal* 13.3 (2005), pp. 227–245.

- [68] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. “Evaluating fuzz testing”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138.
- [69] Bogdan Korel and Janusz Laski. “Dynamic program slicing”. In: *Information Processing Letters* 29.3 (1988), pp. 155–163.
- [70] Joxean Koret and Elias Bachaalany. *The antivirus hacker’s handbook*. John Wiley & Sons, 2015.
- [71] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. “Testing closed-source binary device drivers with DDT”. In: *USENIX Annual Technical Conference*. 2010.
- [72] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. “Genprog: A generic method for automatic software repair”. In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 54–72.
- [73] JongHyup Lee, Thanassis Avgerinos, and David Brumley. “TIE: Principled Reverse Engineering of Types in Binary Programs”. In: *Proceedings of the 18th Annual Network and Distributed System Security Symposium*. 2011.
- [74] Caroline Lemieux and Koushik Sen. “FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. 2018, pp. 475–485.
- [75] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. “Steelix: program-state based binary fuzzing”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 627–637.
- [76] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. “Automatic reverse engineering of data structures from binary execution”. In: *Proceedings of the 17th Annual Network and Distributed System Security Symposium*. 2010.
- [77] Cullen Linn and Saumya Debray. “Obfuscation of executable code to improve resistance to static disassembly”. In: *Proceedings of the 10th ACM conference on Computer and communications security*. 2003, pp. 290–299.
- [78] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: Building customized program analysis tools with dynamic instrumentation”. In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

- [79] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. “Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 389–400.
- [80] Xiaozhu Meng and Barton P Miller. “Binary code is not easy”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 24–35.
- [81] *Microsoft Bounty Program*. <https://www.microsoft.com/en-us/msrc/bounty>. 2018.
- [82] Barton P. Miller, Louis Fredriksen, and Bryan So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44.
- [83] Barton Paul Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Tech. rep. University of Wisconsin-Madison, Computer Sciences Department, 1995.
- [84] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. “BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking”. In: *Proceedings of the 26th USENIX Security Symposium*. 2017.
- [85] Max Moroz and Kostya Serebryany. *Guided in-process fuzzing of Chrome components*. <https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html>. 2016.
- [86] Meinard Müller. *Information retrieval for music and motion*. Springer, 2007.
- [87] Eugene W Myers. “An O(ND) difference algorithm and its variations”. In: *Algorithmica* 1.1-4 (1986), pp. 251–266.
- [88] Vijay Nagarajan, Rajiv Gupta, Xiangyu Zhang, Matias Madou, Bjorn De Sutter, and Koen De Bosschere. “Matching control flow of program versions”. In: *2007 IEEE International Conference on Software Maintenance*. IEEE. 2007, pp. 84–93.
- [89] Vijay Nagarajan, Dennis Jeffrey, Rajiv Gupta, and Neelam Gupta. “A system for debugging via online tracing and dynamic slicing”. In: *Software: Practice and Experience* 42.8 (2012), pp. 995–1014.
- [90] Nicholas Nethercote and Julian Seward. “How to shadow every byte of memory used by a program”. In: *Proceedings of the 3rd international conference on Virtual execution environments*. 2007, pp. 65–74.

-
- [91] Nicholas Nethercote and Julian Seward. “Valgrind: A framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan Notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.
 - [92] James Newsome and Dawn Xiaodong Song. “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software.” In: *Proceedings of the Network and Distributed System Security Symposium*. 2005.
 - [93] Oulu University Secure Programming Group. *A Crash Course to Radamsa*. <https://github.com/aoh/radamsa>.
 - [94] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. “A fistful of red-pills: How to automatically generate procedures to detect CPU emulators”. In: *Proceedings of the 3rd USENIX Workshop on Offensive Technologies*. 2009.
 - [95] Peachtech. *Peach fuzzer platform*. <https://www.peach.tech/products/peach-fuzzer/>. 2018.
 - [96] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. “Cross-architecture bug search in binary executables”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 709–724.
 - [97] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D’Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. “Measuring and Defeating Anti-Instrumentation-Equipped Malware”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2017, pp. 73–96.
 - [98] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. “Argos: An Emulator for Fingerprinting Zero-day Attacks for Advertised Honey-pots with Automatic Signature Generation”. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. ACM, 2006, pp. 15–27.
 - [99] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. <https://llvm.org/docs/LibFuzzer.html>. 2016.
 - [100] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. “LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks”. In: *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06)*. 2006, pp. 135–148.
 - [101] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. “Vuzzer: Application-aware evolutionary fuzzing”. In: *Proceedings of the Network and Distributed System Security Symposium*. 2017.

- [102] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. “Optimizing Seed Selection for Fuzzing”. In: *Proceedings of the 23rd USENIX Security Symposium*. 2014, pp. 861–875.
- [103] Kevin A Roundy and Barton P Miller. “Binary-code obfuscations in prevalent packer tools”. In: *ACM Computing Surveys* 46.1 (2013), p. 4.
- [104] Stan Salvador and Philip Chan. “FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space”. In: *Intelligent Data Analysis* 11.5 (Oct. 2007), pp. 561–580.
- [105] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2005, pp. 263–272.
- [106] Julian Seward and Nicholas Nethercote. “Using Valgrind to Detect Undefined Value Errors with Bit-Precision.” In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 17–30.
- [107] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. “Cardinal Pill Testing of System Virtual Machines.” In: *USENIX Security Symposium*. 2014, pp. 271–285.
- [108] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. “SoK:(state of) the art of war: Offensive techniques in binary analysis”. In: *2016 IEEE Symposium on Security and Privacy*. 2016, pp. 138–157.
- [109] Asia Slowinska and Herbert Bos. “Pointless tainting?: Evaluating the practicality of pointer tainting”. In: *Proceedings of the 4th ACM European Conference on Computer systems*. 2009, pp. 61–74.
- [110] T.F. Smith and M.S. Waterman. “Identification of common molecular subsequences”. In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197.
- [111] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. “BitBlaze: A new approach to computer security via binary analysis”. In: *International Conference on Information Systems Security*. 2008, pp. 1–25.
- [112] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. “Looking inside the black-box: capturing data provenance using dynamic instrumentation”. In: *International Provenance and Annotation Workshop*. 2014, pp. 155–167.

-
- [113] 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: *Proceedings of the Network and Distributed System Security Symposium*. 2016.
 - [114] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
 - [115] “Ukraine power cut ‘was cyber-attack’”. In: *BBC* (Jan. 11, 2017). URL: <https://www.bbc.com/news/technology-38573074>.
 - [116] Nitin H Vaidya and Sohail Hameed. “Scheduling data broadcast in asymmetric communication environments”. In: *Wireless Networks* 5.3 (1999), pp. 171–182.
 - [117] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. “Ifuzzer: An evolutionary interpreter fuzzer using genetic programming”. In: *European Symposium on Research in Computer Security*. 2016, pp. 581–601.
 - [118] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. “Protection of software-based survivability mechanisms”. In: *2001 International Conference on Dependable Systems and Networks*. 2001, pp. 193–202.
 - [119] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. “Skyfire: Data-driven seed generation for fuzzing”. In: *2017 IEEE Symposium on Security and Privacy*. IEEE. 2017, pp. 579–594.
 - [120] Tao Wang and Abhik Roychoudhury. “Using compressed bytecode traces for slicing Java programs”. In: *Proceedings of the 26th International Conference on Software Engineering*. 2004, pp. 512–521.
 - [121] 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*. 2010, pp. 497–512.
 - [122] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. “Analyzing Concurrency Bugs Using Dual Slicing”. In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 253–264.
 - [123] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* 4 (1984), pp. 352–357.
 - [124] Ryan Whelan, Tim Leek, and David Kaeli. “Architecture-Independent Dynamic Information Flow Tracking”. In: *Compiler Construction*. Ed. by Ranjit Jhala and Koen De Bosschere. Springer Berlin Heidelberg, 2013, pp. 144–163.

- [125] Carsten Willems, Thorsten Holz, and Felix Freiling. “Toward automated dynamic malware analysis using CWSandbox”. In: *IEEE Security & Privacy* 5.2 (2007).
- [126] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. “A Survey on Software Fault Localization”. In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.
- [127] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. “Scheduling Black-box Mutational Fuzzing”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. 2013, pp. 511–522.
- [128] Bin Xin, William N. Sumner, and Xiangyu Zhang. “Efficient Program Execution Indexing”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008, pp. 238–248.
- [129] Bin Xin and Xiangyu Zhang. “Efficient online detection of dynamic control dependence”. In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 185–195.
- [130] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2011, pp. 283–294.
- [131] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. “Panorama: Capturing system-wide information flow for malware detection and analysis”. In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 116–127.
- [132] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. “QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing”. In: *27th USENIX Security Symposium*. 2018, pp. 745–761.
- [133] Michal Zalewski. *American Fuzzy Lop*. <http://lcamtuf.coredump.cx/afl/>. 2017.
- [134] Michal Zalewski. *American Fuzzy Lop – README*. <http://lcamtuf.coredump.cx/afl/README.txt>. 2016.
- [135] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. “A study of effectiveness of dynamic slicing in locating real faults”. In: *Empirical Software Engineering* 12.2 (2007), pp. 143–160.
- [136] Xiangyu Zhang, R. Gupta, and Youtao Zhang. “Precise dynamic slicing algorithms”. In: *Proceedings of the 25th International Conference on Software Engineering*. 2003, pp. 319–329.

- [137] Xiangyu Zhang and Rajiv Gupta. “Cost effective dynamic program slicing”. In: *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. 2004, pp. 94–106.
- [138] Xiangyu Zhang and Rajiv Gupta. “Matching execution histories of program versions”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM. 2005, pp. 197–206.
- [139] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. “Experimental Evaluation of Using Dynamic Slices for Fault Location”. In: *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*. AADEBUG’05. ACM, 2005, pp. 33–42.
- [140] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. “TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking”. In: *SIGOPS Operating Systems Review* 45.1 (2011), pp. 142–154.
- [141] *Zynamics BinDiff*. <https://www.zynamics.com/bindiff.html>.

Publications

Papers

The papers associated with this thesis have been removed for copyright reasons. For more details about these see:

<http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-157626>

FACULTY OF SCIENCE AND ENGINEERING

Linköping Studies in Science and Technology, Dissertations, No. 1993, 2019
Department of Computer and Information Science

Linköping University
SE-581 83 Linköping, Sweden

www.liu.se