

ANDREA JEMMETT

UBERFUZZ

Vrije Universiteit Amsterdam



Master Thesis

Uberfuzz: A Cooperative Fuzzing Framework

Author: Andrea Jemmett (2573223)

1st supervisor: prof.dr.ir. H.J. Bos

2nd reader: dr. C. Giuffrida

*A thesis submitted in fulfillment of the requirements for
the Master of Science in Artificial Intelligence*

August 30, 2019

To my parents, who have been always supportive of my choices. To my grandparents, who have patiently waited for the completion of every step I undertook to this moment. To Nensi, who has always been understanding and has always found a way to give me the necessary energy to complete this effort.

ABSTRACT

Fuzzing is a popular technique for testing software for reliability and security. As different fuzzers are specialized to different kinds of software and make different assumptions about it, the practitioner is often tasked to select the appropriate fuzzer for the Software Under Test (SUT). Otherwise, if enough resources are available, they choose a set of fuzzer to run — *independently* — in parallel or sequentially.

In this thesis we present a *Cooperative Fuzzing Framework (CFF)* that allows a set of fuzzers, running in parallel, to communicate and exchange information. We describe a distributed implementation of the framework that uses hardware-generated coverage feedback to control the flow of information among the fuzzer instances. Moreover, the system is designed to integrate with a generic fuzzer that implements an API which is already implemented by most fuzzers.

We evaluate the CFF using four popular fuzzers on four UNIX utilities. The results show promising improvements both in terms of code coverage and unique crashes found.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Herbert Bos, for giving me the opportunity to develop my Master project in the Systems and Network Security Group.

Moreover, I would like to thank all members of the VUsec, in particular Cristiano Giuffrida and Kaveh Razavi, for their help, discussions and stimulating knowledge provided.

CONTENTS

1	INTRODUCTION	1
1.1	Quality Assurance and Control	2
1.2	Developing High-Quality Software	3
1.2.1	Defensive Programming	4
1.2.2	Code Reviews	5
1.2.3	Advancements in Programming Languages	6
1.2.4	Formal Methods	7
1.3	Software Testing	8
1.3.1	Testing Techniques	9
1.3.2	Fuzz Testing	14
1.4	Thesis Outline	17
2	BACKGROUND AND RELATED WORK	19
2.1	Black-Box Mutational Fuzzing	19
2.2	Coverage-Based Gray-Box Fuzzing	21
2.2.1	American Fuzzy Lop	22
2.2.2	Honggfuzz	26
2.2.3	VUzzer	26
2.3	Symbolic-Assisted Fuzzing	29
2.4	Hybrid Techniques	32
2.5	Cooperative Fuzzing	34
3	COOPERATIVE FUZZING FRAMEWORK	37
3.1	System Design	37
3.1.1	Common Fuzzer Interface	37
3.1.2	Central Decisional Unit	39
3.1.3	Cooperative Fuzzing Strategies	41
3.2	System Implementation	43
3.2.1	Communication Channels	43
3.2.2	Driver Implementation	45
3.2.3	Master Implementation	47
4	EVALUATION	49
4.1	Single Fuzzer Evaluation	51

4.2	Cooperative Fuzzing Evaluation	54
4.3	Crash Analysis	57
4.3.1	Known Vulnerabilities	59
4.4	Overhead Evaluation	61
5	DISCUSSION	65
6	FUTURE WORK	69
7	CONCLUSION	71
A	BAYESIAN ESTIMATION OF COOPERATIVE STRATEGIES	73
	BIBLIOGRAPHY	83

LIST OF FIGURES

Figure 1.1	Waterfall model of software development life-cycle	3
Figure 3.1	Cooperative Fuzzing Framework communication model	40
Figure 3.2	Communication channels between fuzzer, driver and master.	44
Figure 4.1	Mean coverage over time for single fuzzers. . .	52
Figure 4.2	Single fuzzers: distribution of difference of means for djpeg.	53
Figure 4.3	Single fuzzers: distribution of difference of means for objdump.	53
Figure 4.4	Single fuzzers: distribution of difference of means for tiff2pdf.	54
Figure 4.5	Distribution of difference of means for union of fuzzers against the best single fuzzer.	55
Figure 4.6	Mean coverage over time for two cooperative strategies and union of fuzzers.	57
Figure 4.7	Unique crashes over time for listswf.	59
Figure 4.8	Density of unique crashes over time for listswf, divided in intersection of hashes and not in the intersection.	60
Figure 4.9	Bugs with assigned CVE identifier found in listswf.	63
Figure A.1	Bayesian estimation for single winner strategy vs. union of fuzzers for djpeg.	73
Figure A.2	Bayesian estimation for single winner strategy vs. multiple winners strategy for djpeg.	74
Figure A.3	Bayesian estimation for single winner strategy vs. union of fuzzers for objdump.	75
Figure A.4	Bayesian estimation for multiple winners strategy vs. union of fuzzers for objdump.	76

Figure A.5	Bayesian estimation for single winner strategy vs. multiple winners strategy for objdump. . . .	77
Figure A.6	Bayesian estimation for multiple winners strategy vs. union of fuzzers for tiff2pdf.	78
Figure A.7	Bayesian estimation for multiple winners strategy vs. single winner strategy for tiff2pdf. . .	79
Figure A.8	Bayesian estimation for multiple winners strategy vs. union of fuzzers for listswf.	80
Figure A.9	Bayesian estimation for multiple winners strategy vs. single winner strategy for listswf. . .	81

LIST OF TABLES

Table 3.1	Cooperative Fuzzing Framework interface with fuzzers	39
Table 4.1	Number of basic blocks and functions for the chosen targets.	51
Table 4.2	Mean coverage with 95% confidence intervals for single fuzzers. Highlighted is the best for the given program.	51
Table 4.3	Mean coverage with 95% confidence intervals for best single fuzzer and union of coverage traces.	54
Table 4.4	Mean coverage with 95% confidence intervals for winning strategies that select single or multiple winners and without cooperation. . . .	56
Table 4.5	Distinct unique crashes and amount discovered by one and not discovered by another. . .	58

LIST OF ALGORITHMS

Algorithm 1	Black-box mutational fuzzing	19
Algorithm 2	Coverage-based Gray-box Fuzzing	21
Algorithm 3	General scheme for an Evolutionary Algorithm	27
Algorithm 4	Symbolic-Assisted Fuzzing	29
Algorithm 5	Generic strategy for the Cooperative Fuzzing Framework	41

LISTINGS

Listing 1.1	Defensive programming: unsafe example . . .	4
Listing 1.2	Defensive programming: safe example	4
Listing 2.1	AFL's instrumentation	23
Listing 4.1	Unchecked memory allocation in <code>util/read.c:222</code> causing CVE-2017-7582.	60

ACRONYMS

API	Application Programming Interface
ASLR	Address Space Layout Randomization
BFF	Basic Fuzzing Framework
BTS	Branch Trace Store
CFE	Cooperative Fuzzing Framework
CFG	Control Flow Graph
CGF	Coverage-based Gray-box Fuzzer
CLR	Common Language Runtime
CVE	Common Vulnerabilities and Exposures
DTA	Dynamic Taint Analysis
EA	Evolutionary Algorithm
FCS	Fuzz Configuration Scheduling
FFI	Foreign Function Interface
HDI	Highest Density Interval
JVM	Java Virtual Machine
MAB	Multi-Armed Bandit
OS	Operating System
PT	Processor Trace
SDL	Security Development Lifecycle
SUT	Software Under Test
XP	Extreme Programming

INTRODUCTION

Software products are becoming essential in our daily lives, from managing the most simple household appliances to sensitive applications such as military, medical and transportation. Producing high-quality software becomes more and more of a necessity and software developers need to aim at it. Crosby [40] defines quality as *conformance to requirements*. Requirements have to be clearly stated to avoid misunderstandings and the development process is constantly monitored to check conformance of the product to those requirements. As an example, one requirement for a web server may be that it must be able to serve at least a thousand concurrent requests within a set amount of time. If the web server fails to do so, the product does not meet its requirements and should be rejected because of poor quality. Notice that requirements may belong to different *quality parameters* such as functionality, usability, reliability, performance, security and so on.

Fuzzing is a popular technique capable of testing software for security and reliability; fuzzers are used extensively by major companies such as Google [1, 37, 44, 86] and Microsoft [13, 18, 64]. As most fuzzing engines are highly specialized to a particular kind of software (e. g. network protocols, operating systems, compilers), the practitioner is often responsible to decide which fuzzer to use for the SUT; alternatively, if enough resources are available, they run different fuzzers sequentially or in parallel.

We propose a framework for interfacing with and coordinating different kinds of fuzzers, running in parallel, with the aim to harness the unique features of each one by means of cooperation. We also describe a possible implementation of such framework which we then evaluate on four UNIX utilities. The results show promising gains

in code coverage and number of unique crashes found, compared to traditional fuzzing setups.

The remainder of this chapter provides an introduction to software quality control and testing. [Chapter 2](#) presents a detailed view of some testing techniques closely related to our work, with an emphasis on the fuzzers used by our system. [Chapter 3](#) presents our framework and proposed implementation; [Chapter 4](#) contains its evaluation.

1.1 QUALITY ASSURANCE AND CONTROL

Quality assurance aims at improving the quality of the product by establishing practices within the organization and training the team; it comprises procedures and activities assuring that the requirements will be fulfilled. Quality control on the other end, refers to activities that enable verification of a product, gathering of statistics and metrics, discovery of defects and ensures that those are fixed before release or passing the intermediate product to the following stage in the development process.

A *failure* manifests itself whenever the system does not behave accordingly to its specification and is caused by a *fault* which in turn is caused by an *error* made by the software engineer. When a failure causes the system to abort its execution unexpectedly, it is called a crash. A fault (also known as a bug) is a specific condition in the system that causes it to behave in unexpected ways. A failure can be caused by a programming error (an error in the source code), a design flaw or even by an external component such as a library, the Operating System (OS) or even the compiler.

Quality assurance prescribes that the whole software development lifecycle is checked for quality as well as the intermediate and final products. [Figure 1.1](#) presents a simplified representation of the classical waterfall model for software development lifecycle. Each stage produces an intermediate artifact that serves as input for the next stage. Quality control provides tools and methodologies to check the quality of artifacts and in case a defect is found, the artifact is rejected (development does not transition to the following phase). In other

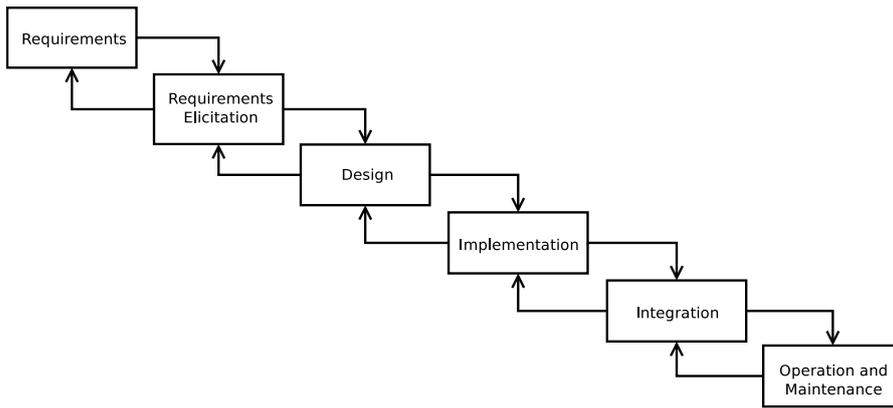


Figure 1.1: Waterfall model of software development lifecycle

words, quality control allows a software engineer to gauge quality of a product at each development stage, preventing the propagation of errors from one stage to the next.

In an effort to minimize the amount of security bugs in their applications, Microsoft engineers have devised a methodology called Security Development Lifecycle (SDL) [78]. SDL presents a development model similar to the waterfall model, where a series of phases is applied in succession with intermediate artifacts and quality checks, but enriches it with established security practices and methods. For each stage of development, SDL provides clear principles, tools and practices to assess quality from a security perspective, allowing for a progressive discovery of problems, throughout the entire development lifecycle.

1.2 DEVELOPING HIGH-QUALITY SOFTWARE

There is a good number of tools and practices at disposal of the software engineer that enable production of high-quality software. Some are suited suited for assessing the quality of artifacts from a static or dynamic perspective; other try to prevent entirely certain classes of errors.

The best way to obtain a quality product is to put it there in the first place. Careful craftsmanship following a set of good practices results in products with less errors. Writing clean code that is readable

```

char* capitalize(char* input)
{
    char* str = (char*) malloc(100 * sizeof(char));
    strcpy(str, input);
    str[0] = toupper(str[0]);
    return str;
}

```

Listing 1.1: Defensive programming: unsafe example

```

char* safe_capitalize(char* input)
{
    if (input == NULL)
        return NULL;
    char* str = (char*) malloc(100 * sizeof(char));
    if (str == NULL)
        return NULL;
    strncpy(str, input, sizeof(str));
    str[sizeof(str) - 1] = 0;
    str[0] = toupper(str[0]);
    return str;
}

```

Listing 1.2: Defensive programming: safe example

and refactored, results in code that is more maintainable and easier to inspect and review [47, 83]. Training the development team and following a set of good practices (for example secure coding) becomes crucial, as also highlighted by the first step in SDL.

1.2.1 *Defensive Programming*

Well trained programmers know the importance of defensive programming, a set of practices including input validation and secure coding. Input validation ensures that data that is processed by the system, coming from an external source, is checked for correctness and security. An high-quality program should never expect data to be valid, even when it comes from a trusted source as reliable software should be resilient to the unexpected. Input validation becomes even more important for applications processing data from untrusted sources such as the Internet [95].

[Listing 1.1](#) and [Listing 1.2](#) present two versions of a function to capitalize strings. The former can behave unexpectedly in a number of situations: the input string can be a null-pointer causing the program to crash inside the `strcpy` function; the `malloc` function may be unable to allocate memory, returning a null-pointer resulting in a similar crash inside `strcpy`; the input string can be longer than the allocated buffer, causing the program to crash again inside `strcpy`; ultimately, the resulting string may not be null-terminated, possibly causing a memory leak in other parts of the program. There are four possible failures waiting to happen in only four lines of code. Defensive programming practices aim at preventing those kind of errors, as shown in [Listing 1.2](#).

1.2.2 Code Reviews

As software developers can fail to follow defensive programming practices (for example due to high pressure to finish a software module), some errors might still slip into the code. Code review can help find errors in the source code early on. In the typical scenario, one or more developers, not including the author, visually inspect the source code (of a module, function or entire program) with the explicit purpose of finding programming errors or ways to improve (the quality of) the code. Code review assumes a central role in companies such as Microsoft [8], Facebook [45] and Google [68] and is often facilitated by Web-based collaboration tools. Code reviews are taken to the extreme—as in Extreme Programming (XP) [14]—by a practice called pair programming. In this practice, two developers produce source code in close collaboration: the “driver” has control of the keyboard and writes code, while the other (also called “observer” or “navigator”) watches over the driver’s work as it is typed, trying to spot errors, proposing alternatives and considering strategic implications on future work. The two roles are switched periodically to maintain equally shared ownership of the product.

1.2.3 *Advancements in Programming Languages*

Most performance critical software applications are developed in languages like C [69] or C++ [102] as their constructs allow developers to heavily optimize their code for specific platforms. These languages, while providing an higher level of abstraction over machine code or assembly languages, maintain a set of low-level features (such as direct memory access and pointer arithmetics) that can be easily misused, ultimately inducing programming errors. Careless use of pointer arithmetic is the major cause of a class of bugs (known as memory corruption bugs) such as buffer overflows, null-pointer dereferences, use-after-frees and double frees.

Programming languages like Java [55] and C# [58], instead, provide automatic memory management: allocation on the heap is hidden by object creation and deallocation is done periodically by the garbage collector. Moreover, programs written in such languages, are not compiled directly to machine code, instead are compiled to an intermediate artifact which in turn is interpreted by a virtual machine; Java runs on the Java Virtual Machine (JVM) while C# runs on the Common Language Runtime (CLR). This improves security and reliability, as programs run in a controlled environment, but may drastically degrade performance; besides virtual machines are generally written in highly optimizable languages such as C or C++ and can still be susceptible to memory corruption bugs.

Recent advancements in programming languages try to guarantee memory safety while still producing high-performance binaries. Rust [84] uses the concept of ownership of data to enable the compiler to make memory safety guarantees without the need of a garbage collector. A type system can be useful to rule out a series of programming errors as there is a guarantee that inappropriate arguments will not be applied to an operation [35]. This check can be done statically by the compiler, in a process called typechecking, or dynamically at run-time. Typeful programming [34] is a style of programming where types are pervasive; it is central to the design of languages such as Haskell [65] (e. g. the concepts of kinds and type constructors). De-

pendent types [6] augment the expressiveness of a type system by allowing the definition of types with logical predicates over values. As an example, a function's return type may depend on the value of the argument, not just its type (as with polymorphism and generic programming). In other words, a function accepting a positive integer may return a list with precisely that number of elements: this specification can be encoded with a dependent type and checked by the typechecker. Early implementations of programming languages with dependent types are Dependent ML [112, 113] and Cayenne [7]; more recently Idris [28] and F* [103].

1.2.4 *Formal Methods*

Formal methods are a set of mathematical techniques that enable verification of software (and hardware) engineering artifacts. In a strict sense, they can be used to prove with absolute certainty that an artifact conforms to its specification [57]. For example they can be used at the design level (e. g. requirement specification, algorithm design), where properties of a formal specification are proved through model-checking or theorem-proving techniques. Formal methods can also be used at the source code level to prove that a program (or function or module) complies with its specification; more concretely, given a set of preconditions, program execution is proven to adhere to certain postconditions. For this to be possible, programming languages have to allow encoding the precise semantics of a program.

As formal methods require significantly more effort, are not applicable to all programs and do not scale well, they are often used only for critical applications such as aerospace, financial systems and defence [111]. Recent advancements have allowed for the development of more complex software, tackling the scaling problem. An example is an OS kernel formally verified from its specification to its implementation [71].

1.3 SOFTWARE TESTING

Section 1.2 presented techniques for developing high-quality software that fall under the category of *verification and validation* (V&V). The IEEE process standard [63] defines verification and validation as the processes used to establish that the product and any intermediate artifact conforms to its requirements and fits its intended use. As Boehm put it [19]: verification answers the question of “Am I building the product right?”; validation instead answers the question of “Am I building the right product?”. Verification and validation are built into the software development lifecycle and help the software engineer gauging the quality of its product. Given this definition, it is easy to see how verification and validation is a further categorization of software quality control.

Software verification can be applied by means of two approaches: static or dynamic [48]. Static verification inspects the product without executing it; code reviews, compiler warnings, programming language style checkers and formal methods are all examples of static verification. With dynamic verification (commonly referred to as software testing) instead, the software is executed and its behaviour is inspected. Because dynamic verification can be applied with varying granularity, it can be employed at any moment during the development lifecycle and not only at its end, when the product is finished.

Software testing is defined in a concise way in [99], with emphasis on key aspects:

Software testing consists of the *dynamic* verification that a program provides *expected* behaviours on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain.

Software testing is an inherently dynamic practice as the SUT needs to be inspected in an environment that most closely resembles the production deployment environment (in which users directly interact with the product). Software testing complements static verification as it allows to verify the interaction of the SUT with its environment

(e. g. hardware, networking, OS) [5]. The state of the SUT itself, as well as that of its environment (where it can affect the SUT's output), are considered part of the input. This makes the domain of all possible inputs spike in size, even for trivial programs; because of this the number of test cases has to be finite and the software tester should not aim to test for all possible inputs. Because even a trivial program can have a virtually infinite input domain, software testing cannot verify a program completely [67]. Follows that testing can reveal the presence of bugs, not their absence [31, 43].

As the software tester is required to work with a finite set of test cases, selecting the subset of inputs from the domain of all possible inputs to the SUT is a key aspect in software testing. Many testing techniques differ especially in how test cases are selected [87] and based on the specific nature of the SUT (e. g. its domain, design or implementation details) different selection techniques may yield greatly different results in terms of testing effectiveness.

A test case is of any value to the testing process if, for the selected input to the SUT, there is an associated outcome that the software tester expects to observe in response. Examples of outcomes are: output values produced by the program, state changes within the program or on external entities and a composition of a set of smaller outcomes [87]. Test oracle is a term frequently used in software testing to refer to an entity capable of telling the expected outcome of a specific test case [62]. In its most general form a test oracle can be thought of as a predicate that tells us whether the observed outcome is acceptable or not [11]. A test case is indeed defined by the input to the SUT and an associated test oracle.

1.3.1 Testing Techniques

As already stated, software testing techniques differ from each other the most when compared along the axis of how they perform test case generation. The major distinction is given by the kind of source of information the software tester has access to during test design. Testing approaches are commonly classified into *white box* or *black box*: in

the former (also called structural testing) sources revealing the inner structure of the SUT are used (e. g. source code), with a focus on data and control flow features; in the latter (also called functional testing) no internal detail about the SUT is known and only the front-facing functionalities are tested, as the sole source of information available is the product specification.

Control and data flow features are used to reason about implementation details of the SUT. Control flow refers to the way instructions in a program are executed. The processor passes control from one instruction to another in different ways, in the most common case one instruction follows another, but control can be passed also by means of function calls, interrupts, message passing or conditional statements. Data flow refers instead to the way values are propagated within the program and represents declaration and usage of variables in the source code.

As white and black box testing look at the SUT from the point of view of the developer and the end user respectively, they complement each other to achieve a complete testing effort. While white box testing provides tools to uncover errors pertaining control or data flow, it does not provide a way to expose high-level functional flaws or gauge quality in regards of usability or another front-facing factor. The opposite is true for black box testing. The middle ground of both practices is an approach called intuitively gray box testing.

Gray box testing uses little, partial or inferred information about the structure of the SUT to enhance or focus black box testing. Knowledge about the algorithms being used, architectures, the operating environment in which the SUT is going to interact or high-level descriptions of its behaviour, may be taken into consideration when designing test cases. Moreover, gray box testing approaches interact with the SUT only via its front-facing functionalities (those exposed to the end user), similarly to black box testing. Gray box testing is particularly suited, for example, for Web application testing [42, 88] as Web applications are made up of an high number of components and having knowledge of how they interact with each other is vital to the effectiveness of testing.

The most basic source of information for test case generation comes from the software engineer's expertise, own intuition and experience with similar applications. This kind of *ad hoc* testing is useful to produce highly specialized test cases, where other (more formal) testing approaches may fail to do so. Ad hoc testing is not structured; *exploratory testing* [66, 94] enhances it with a methodological approach: test-related learning, design, execution and interpretation of test results are activities that must be run in parallel with each other. Each of these activities is mutually supportive and it is the software tester's responsibility to manage her own resources by choosing a combination of activities that best optimize the value of her work. In exploratory testing, the testing process itself, through inspection of the program behaviour, is used as a source of information to make decisions about further testing. Exploratory testing is an inherently iterative process.

The SUT's input and output domains are another fundamental source of information that a software engineer can draw from to design test cases. In this context, the term *test vector* represents the set of values that form the input of a test case; values are drawn from the input domain of the considered input variable. Generating test cases from the input domains consists in computing an expected output for selected input values; this often results in a large number of test cases as every combination of special values of the input variables needs to be represented by a test vector. The advantage is that generating the expected output for a given test vector is generally trivial because the SUT's specification can be directly used. In contrast, generating test cases from the analysis of the output domains can produce less test cases (as there is no need to consider different combinations of special values of the output domains) at the cost of an increased difficulty in generating the test vector capable of producing the desired output.

A solution to reduce the number of test cases when using an approach based on the analysis of the input domains is to use *pairwise testing* [41, 82, 106]. In contrast to exhaustive testing, where all combinations of all input variables' special values have to be represented by a test vector, pairwise testing considers such combinations only for pairs of variables. Pairwise testing can be generalized to a t number

of variables, instead of pairs, also referred to as *t-wise* testing. The resulting set of test cases is a subset of the one produced by exhaustive testing; nonetheless pairwise testing has been proven effective when properly applied [9, 74]. As generating the minimum amount of test cases for pairwise testing is an NP-complete problem [76], a considerable number of strategies have been proposed [56] such as orthogonal arrays and in parameter order.

Another popular approach based on the analysis on the input domain is *equivalence class partitioning* [93]. As the name suggests it involves partitioning the input domain into subsets (or equivalence classes) based on a specified criterion and then use each of these subsets as a source for at least one test case. The subsets are defined such that test inputs drawn from the same equivalence class exercise a similar behaviour on the SUT. Equivalence class partitioning is often applied as a black box technique, by using the SUT's functional specification to derive the partitioning criterion and operating on its interface; however it can also be used as a gray box technique to take into account for control and data flow features.

Boundary-value analysis consists in selecting test values near or on the boundary of a data domain so that test both from inside and outside of an equivalence class are considered. The rationale behind this technique is that boundary conditions are often overlooked or poorly implemented by designers and developers, so including test cases that test these boundary conditions is considered good practice.

Random testing constitutes an apt choice to evaluate the reliability of software systems. Initially applied to the evaluation of hardware systems [29], random testing consists in independently sampling inputs from the input domain, executing the SUT on the selected input and compare the computed result with the expected result (using for example a test oracle) [15]. Knowledge about the input domain, the SUT internals or its functional specification can be exploited to guide the sampling algorithm (e. g. [38]); because of this, random testing provides a relatively simple basis for more complex gray box automated testing techniques, as the next chapter shows.

Access to the source code, knowledge of the internal structure or other inferred control-flow information is at the core of *code-based testing* techniques. The first step in most techniques entails the construction of a Control Flow Graph (CFG) [2, 3]. Each node in a CFG represents a basic block which is a sequence of instructions without jump targets or jumps between them. A jump target would mean the start of a basic block while a jump signals the end of a basic block. Directed edges connecting nodes of the graph represent transfer of control within the program. The aim of code-based testing is producing a corpus of test cases able to cover as many basic blocks, branches or execution paths as possible. As even for trivial programs, due to loops, the number of executable paths can be intractable, the software tester is often required to select a subset of paths from the CFG to test for. Common path selection criteria are statement and branch coverage which aim at executing all basic blocks or edges of the CFG at least once.

After the execution paths have been selected, the objective is to find inputs to the SUT capable of forcing execution of such paths. This is often done via *symbolic substitution*, a process in which constraints exercised by the basic blocks along a selected path are expressed uniquely in terms of the input vector. These constraints are then solved in order to obtain concrete values for the input vector. The process of collecting path constraints can be automated through *symbolic execution* of the program [27, 39, 70]. An interpreter reads and simulates execution of the program, assuming symbolic values for the inputs instead of reading concrete ones (e. g. reading from a file, getting arguments to a unit of code). The interpreter stores the state of the execution and collects path constraints whenever a branching point is reached in the code. There, execution is forked and the path constraint at that point is logically negated so that the forked state can continue exploration of the other branch. Symbolic execution can be used to find inputs that exercise a selected path or to explore all paths of a program. In the second scenario, follows from the description of its abstract mechanics, that symbolic execution suffers from a *path explosion* problem when tackling large programs or unbounded loops. The latter cause

can be alleviated by putting a bound to unbounded loops, resulting in an under-approximation of the SUT. In general, the path explosion problem can be mitigated by merging similar paths [75], optimize the searching algorithm with heuristics [30, 80] or parallelize execution of independent paths [100].

Another bottleneck for the effectiveness of symbolic execution techniques is constituted by the constraint solver used. Most of the computational resources are used to resolve constraints. Even a small improvement in performance of the constraint solver, results in noticeable gains for the symbolic execution process as a whole. Moreover the constraint solver may have technical limitations on its own (e. g. being able to work only on linear constraints). To alleviate this and other problems (related to interpretation, such as tracking of array indices and pointers or dealing with external code segments and the environment), *concolic execution* was proposed. It uses information from the concrete execution of the SUT to simplify symbolic execution where necessary. The SUT runs normally with some concrete input and symbolic execution runs in parallel, providing concrete values to allow concrete execution to explore different branches. When the constraint solver finds a path constraint that it cannot solve, concrete values from the concrete execution state are plugged in the constraint so that the theorem prover can work with a simplified version. Korel first proposed the use of concrete execution to aid symbolic execution to generate test vectors that exercise a particular path [72]. More recently concolic execution was proved effective in exploring large and complex C programs [51, 97], multithreaded Java applications [96] and to extend fuzz testing to uncover security vulnerabilities in x86 binaries [49, 52].

1.3.2 Fuzz Testing

Fuzz testing (or fuzzing) is a software testing technique that builds upon random testing. It was originally conceived for reliability testing, but has recently found application in security testing. The term fuzzing dates back to 1988 from a course project for an Advanced Op-

erating Systems class taught at the University of Wisconsin by Barton Miller [105]. After noticing how a thunderstorm was able to scramble his input through a remote terminal and crash the program on the other end, he decided to propose a project to his students to experiment with random testing of UNIX utilities. A group of two students succeeded at the given task and two years later published their findings [85]. Random testing was already in use since the 1950s, when programmers would use punched cards from the trash or decks of random numbers as input to programs [108].

In general, the operation of a fuzzer is similar to random testing: a fuzzer generates some input (often based on some program-specific knowledge) and feeds it to the SUT. The program's execution is then monitored for unexpected behaviours, specifically crashes or timeouts. The process is then reiterated until a stopping criteria is met (e. g. time budget is exhausted, code coverage is reached, defects are observed). Fuzzers can be categorized along three different properties:

- if new samples are generated by changing existing samples (e. g. from a seeding corpus or from previous iterations) or from scratch;
- if it is aware or not of the input structure and relations among its parts (e. g. checksums, protocol-specific values);
- how much it is aware and makes use of the inner structure of the SUT to drive test generation.

Mutation-based fuzzers need a tests corpus in order to seed the first iteration of fuzzing. From this corpus, selected inputs are systematically mutated to produce new inputs that are added to a data store the fuzzer handles. The updated set of inputs is then used in successive fuzzing iterations. In order to improve fuzzing efficiency, Rebert et al. propose a number of algorithms that optimize seed selection for mutation-based fuzzers [92]. Generation-based fuzzers on the other hand, generate inputs from scratch, often with the aid of some kind of model of the SUT's input space (e. g. a grammar to describe the input model [50]).

A fuzzer's awareness of the SUT's input structure can be exploited to generate, with much higher probability, valid or semi-valid inputs. This is important as invalid inputs tend to stress only the parsing components of a program and leave the core business components unexplored. Having an input model (e. g. a formal grammar, a formalized protocol specification) can be extremely beneficial for fuzzing implementations of complex (stateful) protocols or file formats [10, 90]. Unfortunately an input model is not easily (or at all in case of proprietary file formats or protocols) available. In these cases, if enough samples from the input space are available, one could try to infer the underlying input model [12, 54]. Other efforts have been made to detect and repair checksums in randomly generated inputs [107].

As with any other software testing technique, fuzzers can be classified as black box [59–61, 110], white box [53, 101] or gray box [21, 22, 114], depending on the amount of knowledge of the internal structure of the SUT they leverage. While a white box fuzzer may explore deeper program compartments, the resources necessary for program analysis may become prohibitive. Black box approaches on the contrary, are able to generate new inputs very quickly. When aiming for efficiency instead of effectiveness (e. g. when trying to find the maximal number of defects in limited time or to show in minimal time the correctness of the SUT for a percentage of the input space), has been proven that there exists a bound on the time that systematic white box testing can take for each test execution after which black box testing becomes more efficient [20]. Gray box fuzzers implement lightweight program analysis through instrumentation of the SUT (e. g. by injecting snippets of monitoring code directly into the source code, compiled binary or interpreter), making input generation faster than traditional white box approaches while obtaining feedback from the concrete executions (e. g. through code coverage information).

Because fuzzing for bugs can be seen as a search problem, we want to investigate fuzzing in the context of the No Free Lunch Theorem for optimization [109], which states that there is no best search algorithm when the performance is averaged across all possible problems. In the context of fuzzing, we want to examine if one fuzzer performs

decisively better than the others across a diverse set of practical programs. To push this further, we devise a system that tries to harness the power a set of *existing* fuzzers that run in parallel and evaluate its effect. We achieve this thanks to a cooperative framework inspired by cooperative co-evolution in the field of Evolutionary Computing.

1.4 THESIS OUTLINE

This chapter presented some introductory concepts about software quality and software testing. In [Chapter 2](#) we present a more detailed view of different kind of fuzzers (with an emphasis on those used in the implementation of our system) and related work on cooperative fuzzing. [Chapter 3](#) presents the CFF from its design to its implementation, with arguments for the different design decisions we take. [Chapter 4](#) shows results of the evaluation of single fuzzers, run without cooperation, to asses the validity of the No Free Lunch Theorem on our programs sample; we also evaluate the effectiveness in terms of coverage and crashes of the implemented system and cooperation. [Chapter 5](#) includes a discussion on the results of the evaluation before, in [Chapter 6](#), presenting possible improvements. [Chapter 7](#) concludes this work.

BACKGROUND AND RELATED WORK

In this chapter we present the mechanics of different kinds of fuzzers with an emphasis on specific implementations of Coverage-based Gray-box Fuzzers (CGFs) — which are later used to evaluate our Cooperative Fuzzing Framework (CFF). Then we move on presenting efforts of researchers trying to combine different fuzzing engines or testing techniques with the aim of improving performance or efficiency.

2.1 BLACK-BOX MUTATIONAL FUZZING

Black-box *mutational* fuzzing (or *Random Testing*) is a simple testing technique that uses mutation operators on a sample input to produce a new input; a corpus of *valid* files (the more the better) is required to achieve good efficiency by reducing the search space. A simple algorithmic representation is given in [Algorithm 1](#).

Algorithm 1: Black-box mutational fuzzing

Input : Set of samples S
Output: Set of crashing inputs C

```

1  $C \leftarrow \emptyset$ 
2 while stop condition is not met do
3    $t \leftarrow \text{SelectSeed}(S)$ 
4    $t' \leftarrow \text{MutateSeed}(t)$ 
5   if  $t'$  crashes program then
6      $C \leftarrow C \cup \{t'\}$ 
7   end
8 end

```

The main algorithm works within a loop that stops at a predetermined condition such as the end of a time budget, after the first crash has been found or after a certain number of inputs have been tested, to name a few. More rudimentary tools such as **zzuf** [60] or **Radamsa** [59] allow for the tester to apply his or her own stop criteria

for the fuzzing campaign. The `SelectSeed` function on line 3 of [Algorithm 1](#) selects one input from the seeds corpus using the strategy of choice (e.g. stochastically, by execution time, crash density). Next, the `MutateSeed` function applies a mutation operator (e.g. bit-flips, insertion or deletion of words) to the selected input to create a new one which is then fed to the SUT. Different mechanisms can then be deployed to identify whether the program crashed under the given input, often program or OS specific; if the SUT exhibits a fault, the test case is stored with useful information about the occurred fault to later report about it to the tester.

Tools like `zzuf` or `Radamsa` only implement the `MutateSeed` function on line 4, leaving the remaining components' implementation to the tester. For example `zzuf` applies random bit-flips to its input using a set *mutation ratio* (how much of the input to change) fully configurable by the user within an interval or fixed. `Radamsa` performs instead a number of more sophisticated mutation operators such as insert, repeat, drop and swap on entities like bytes, ASCII and Unicode texts or arithmetic manipulations. More complex black-box mutational fuzzers implement all components of [Algorithm 1](#) and are able to exploit knowledge of the running campaign to achieve better results. The **Basic Fuzzing Framework (BFF)** [61] uses *crash density* as a metric to decide which pair of seed input and mutation ratio to use for the next call to `MutateSeed` (BFF uses indeed `zzuf` within its mutation engine). Crash density of a seed is defined as the number of crashes found by fuzzing that seed, divided by the number of total test cases generated by the seed. Each execution of a mutated seed is modeled as a Bernoulli trial where the outcome is whether or not the SUT exhibited a defect. The Binomial distribution that would result from successive trials is approximated by a Poisson distribution as the number of trials is much higher than the number of times a fault is found. The upper bound of the 95% confidence interval of that distribution is then used to compute the probability p_i of selecting the seed file t_i . The same process is applied for a single seed file and a fixed set of mutation ratio ranges so that for each seed file there is a probability distribution over the set of ranges. Woo et al.

give the name Fuzz Configuration Scheduling (FCS) [110] to describe the problem of selecting the next seed and mutation ratio pair to fuzz (what they call a *fuzzing configuration*) and recognize the Multi-Armed Bandit (MAB) [17] nature of the problem. The authors take one step further by modeling black-box mutational fuzzing as a weighted version of the Coupon Collector’s Problem and use those insights to inspect the FCS problem along three different axes that allows them to compose and evaluate a total of 26 MAB algorithms.

2.2 COVERAGE-BASED GRAY-BOX FUZZING

A CGF uses lightweight instrumentation and monitoring of the SUT to gain coverage information. This information is then exploited to provide a solution to the FCS problem [21, 22, 77, 117]. The general approach is described in [Algorithm 2](#).

Algorithm 2: Coverage-based Gray-box Fuzzing

Input : Set of seed inputs S
Output: Set of crashing inputs C

```

1  $C \leftarrow \emptyset$ 
2  $Q \leftarrow S$ 
3 if  $Q = \emptyset$  then
4   |  $Q \leftarrow \{\text{empty file}\}$ 
5 end
6 repeat
7   |  $t \leftarrow \text{SelectNext}(Q)$ 
8   |  $p \leftarrow \text{AssignEnergy}(t)$ 
9   | for  $i \in [0 \dots p]$  do
10  | |  $t' \leftarrow \text{MutateInput}(t)$ 
11  | | if  $t'$  crashes then
12  | | |  $C \leftarrow C \cup \{t'\}$ 
13  | | else if  $\text{IsInteresting}(t')$  then
14  | | |  $Q \leftarrow Q \cup \{t'\}$ 
15  | | end
16  | end
17 until timeout reached or abort signal received

```

The first difference from black-box mutational fuzzing is that a CGF does not need a corpus of seed files to work properly (although it would be more efficient). Zalewski, the author of AFL, was able to

generate valid JPEG images starting from an empty file [116]. The functions `SelectNext` and `MutateInput` (at lines 7 and 10 respectively) are analogues of `SelectSeed` and `MutateSeed` of Algorithm 1. The framework of Algorithm 2 incorporates explicitly the mechanics that *smart* black-box mutational fuzzers like BFF implement. The function `AssignEnergy` at line 8 decides how much effort should be put in fuzzing the selected test-case (e. g. how many mutated inputs should be created from it). Another difference from black-box mutational fuzzers is the `IsInteresting` function at line 13, responsible to determine whether the mutated input is deemed *interesting* and worth fuzzing; this allows for CGFs to build a corpus of test-cases that could even be reused with other tools or to fuzz another software that accepts the same file format. For CGFs, *interesting*, loosely means that increases code coverage and by keeping a queue of test-cases with ever-increasing coverage helps fuzzers of this kind reaching deeper portions of the SUT compared to black-box mutational fuzzers.

2.2.1 American Fuzzy Lop

AFL [114] is one of the most well known CGFs. Its focus is not on any singular principle or insights but is rather a collection of hacks that have been tested and proved effective in practice; the governing principles for its development are speed, reliability and ease of use [117]. AFL gets its coverage feedback from the SUT by instrumenting its compiled binary. This is done by injecting specific locations of the SUT with a monitoring snippet of code. This code can be injected by compiling the SUT with the `afl-gcc` utility or, when the source code is not available, AFL uses QEMU [16] to dynamically (during interpretation, at runtime) instrument the binary. AFL-dyninst [4] is an extension to AFL that injects the instrumentation snippet directly into the binary. What AFL injects into the SUT is essentially equivalent to that presented in Listing 2.1.

The instrumentation snippet is injected at every branch within the instrumented code. The `cur_location` variable is generated randomly at compile time and identifies the current *basic block* (a straight code

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Listing 2.1: AFL’s instrumentation

sequence without branches besides at its entry and exit points). The `shared_mem` array is a 64KB shared memory region provided by the fuzzer; each byte of the shared memory can be thought of as a hit to a transition from one branch to another. The shift operation at the last line preserves directionality of the tuples (e.g. $A \oplus B$ would be indistinguishable from $B \oplus A$) and to keep the identity of loops within the same basic block (e.g. $A \oplus A$ would be equal to $B \oplus B$).

With regards to [Algorithm 2](#), AFL implements the `SelectNext` function by classifying elements of the queue as *favorites*. A test-case is deemed favorite if it exhibits faster speed of execution and small size for the branch tuples that it covers compared to the other test-cases in the queue. AFL selects favorite items more often, implementing a strategy that favors, using the MAB terminology, exploitation against exploration [21]. AFL initially applies a set of deterministic mutation operators to selected inputs and later uses what it calls an *havoc stage* where more complex and stacked mutations are stochastically applied; the function `AssignEnergy` determines how many mutations should be applied to the selected test-case during the havoc stage. AFL’s implementation of `AssignEnergy` uses a mix of execution speed, coverage information and age of the selected test-case to determine how many inputs should be generated by mutating it. AFL implements a good number of mutation operators such as bit-flips, insertion and deletion of bytes, arithmetic operators, to name a few [115]. The function `IsInteresting` as implemented by AFL, returns true if the input t' exercises a new basic block transition or if the number of times a transition is hit goes from one range of values to the next; AFL employs a “bucketing” scheme where the range of values required to be in the next bucket roughly doubles (the exact values are 1, 2, 3, 4 – 7, 8 – 15, 16 – 31, 32 – 127 and 128+) [117].

2.2.1.1 AFLFast

Böhme, Pham, and Roychoudhury model Coverage-based Gray-Box Fuzzing as the systematic exploration of the state space of a Markov chain, where the state space is composed by all the possible paths that the SUT can take [21]. This allows them to conduct a mathematical analysis of the challenges faced by CGFs such as AFL. By observing that an high proportion of test-cases generated by AFL exercise a small number of *high frequency paths* (i. e. states of the Markov chain that are visited more often), the authors draw the intuition that steering fuzzing more toward *low frequency paths* could lead to an improvement in performance (exploring more paths with the same amount of fuzz). This intuition is implemented into AFLFast, an extension of AFL. In reference to [Algorithm 2](#), AFLFast replaces the `SelectNext` and the `AssignEnergy` functions with new implementations.

The authors discuss and evaluate different power schedules, responsible of assigning energy to a selected test-case t_i exercising path i . Besides two of them (which use a constant schedule), the other power schedules are functions of two values: $s(i)$, the number of times t_i has already been selected from the queue and $f(i)$, the number of generated inputs that exercise path i . The former allows for power schedules that assign more energy the more the test-case is selected; the latter serves as an approximation of the stationary distribution of the Markov chain and allows for power schedules that assign energy inversely proportional to the density, effectively directing more fuzzing efforts towards low density regions of the distribution. AFLFast implements nine power schedules: two constant schedules (one of which is the same used by AFL) that assign the same amount of energy every time the test-case is picked from the queue; seven monotonous schedules that assign an increasing amount of energy every time the test-case is picked from the queue.

The same two metrics used in the monotonous power schedules are also used to devise two search strategies: one prioritizes inputs with small $s(i)$ (i. e. test-cases that have not been selected often), one prioritizes inputs with small $f(i)$ (i. e. inputs that exercise a low frequency

path). These strategies are not mutually exclusive and can be used together: first is searched the input with the smallest $s(i)$, if more than one is found it proceeds searching for an input with the smallest $f(i)$; if more than one exists, AFLFast falls back to AFL’s strategy (based on input size and execution time).

The presented evaluation of AFLFast over common UNIX utilities shows that the exponential schedule works best, assigning energy inversely proportional to $f(i)$ and directly proportional to $2^{s(i)}$. The comparison of search strategies shows that the combination of both strategies outperforms significantly both strategies individually and AFL’s strategy. AFLFast shows promising results when directly compared with AFL. When both AFL and AFLFast are able to discover (within the given time budget) the same vulnerability, AFLFast does it much quicker; at the same time it is able to expose vulnerabilities that AFL could not discover.

2.2.1.2 FairFuzz

Lemieux and Sen devise another extension to AFL that targets *rare branches* with the aim to drive the fuzzing process deeper into the SUT [77]. FairFuzz changes how the `SelectNext` and `MutateInput` functions of [Algorithm 2](#) work. When selecting inputs from the queue, FairFuzz prioritizes test-cases that exercise a rare branch; successively new fuzz is produced by applying byte-level mutation operators to the selected input trying to exercise the same rare branches while still exploring new parts of the SUT. The intuition behind this is that for most file formats, there are sequences of bytes acting as headers to prove the validity of the file format and other bytes that trigger execution of various parts of the program that processes the file. Among all the fuzz generated by AFL, only a few will contain the right sequence of bytes in the right place; FairFuzz may classify branches triggered by this header as rare and apply mutation operators that preserve the header in the generated fuzz. More specifically a branch is rare if it is hit by a number of inputs (amount of fuzz) smaller than the *rarity cutoff*. This threshold is computed after each call to `SelectNext` as 2^i where $2^{i-1} < \min(B_{hit}) \leq 2^i$ and B_{hit} is the set of hit

counts for all branches discovered so far. At the core of the mutation operators instead, the authors describe the concept of *branch mask*, an artifact used to determine at which positions in the input bytes can be overridden, deleted or inserted. The branch mask, computed during the deterministic stage of mutation, is then used to steer mutation in the havoc stage so that rare branches are still covered by the mutated input.

2.2.2 *Honggfuzz*

Honggfuzz [104] is another general-purpose CGF which simplifies the semantics of Algorithm 2 but offers state-of-the-art implementation that grants huge throughput (generates lots of fuzz, especially in persistent mode). The most peculiar feature of Honggfuzz is allowing the user to gather feedback from the SUT via software or hardware sources; it supports CPU branch and instruction counting by means of Intel Branch Trace Store (BTS) or Intel Processor Trace (PT) on supported CPUs. Honggfuzz selects uniformly at random one input to mutate from the queue and assigns constant energy of 1 to it. The `MutateInput` function picks uniformly at random one of the implemented mangling functions and applies it to the selected input. Honggfuzz considers a new input interesting if the used coverage metric increases; if using Intel BTS for example, Honggfuzz maintains a bitmap containing branch coverage information (similar to the one used by AFL) and any fuzz exercising a previously unseen branch is considered interesting and added to the queue.

2.2.3 *VUzzer*

VUzzer [91] is an *evolutionary* gray-box fuzzer that uses lightweight static and dynamic analysis to gain knowledge about control- and data-flow features of the SUT, which is then exploited to deploy an *application-aware* mutation strategy. VUzzer follows the same population-

based model of Evolutionary Algorithms (EAs) for which a generic representation is given in [Algorithm 3](#).

Algorithm 3: General scheme for an Evolutionary Algorithm

```

1 Population ← Initialize
2 E ← Evaluate(Population)
3 repeat
4   Parents ← SelectParents(Population, E)
5   Offspring ← Recombine(Parents)
6   M ← Mutate(Offspring)
7   E ← Evaluate(M)
8   Population ← SelectSurvivors(M, E)
9 until Stop criteria are met

```

After the population is initialized it is evaluated using a *fitness function*. The population then undergoes the iterative evolutionary process composed of three phases:

- **parent selection:** a set of parents is selected from the population pool using the chosen strategy and the fitness scores (typically the strategy is probabilistic prioritizing individuals with better fitness score);
- **variation operators:** there are two kinds of variation operators typically used in EAs: recombination operators take two or more parents and combines them to produce an offspring; mutation operators instead are unary;
- **survivor selection:** the offspring is evaluated and selection techniques similar to the ones used in parent selection are applied to select the best individuals from the offspring to generate the population pool for the next iteration.

VUzzer uses static and dynamic analysis on the SUT's binary to deploy enhanced application aware mutation operators and derive a fitness function that uses control-flow features of the binary alongside code coverage. Before starting the main fuzzing loop, VUzzer uses an intra procedural static analyzer which objective is two-fold (i) enumerate all immediate values from `cmp` instructions (ii) for each

function, compute the CFG and model it as a Markov Chain such that to each basic block is assigned a *weight* that is the inverse of the probability of reaching it. The set of immediate values is what the application might expect as input at certain offsets and VUzzer uses these values to mutate the offspring in the main fuzzing loop. The set of weights is instead used alongside a set of *error-handling basic blocks* to compute the fitness score. In contrast to other CGFs like AFL or Honggfuzz, VUzzer requires a minimal set of valid inputs to initialize the set of error-handling basic blocks, which is then incrementally developed during fuzzing. After collecting the set of basic blocks exercised by these initial inputs, VUzzer proceeds at throwing totally random inputs at the SUT and marks basic blocks that were not executed by any of the valid inputs as error-handling basic blocks. This knowledge is then used to penalize individuals that execute such basic blocks, which is done by properly setting the weight for the corresponding basic block. VUzzer uses Intel’s Pin [79] to dynamically instrument the binary and gather the trace of basic blocks exercised by the SUT under a certain input.

Dynamic binary instrumentation is not the only dynamic technique used by VUzzer. Dynamic Taint Analysis (DTA) allows to monitor a *tainted* input within the SUT and gather information about code that operates on it. VUzzer uses DTA to monitor `cmp` and `lea` instructions that are tainted by the input: from the former it extracts the set of offsets in the input that taint the operands (with byte-level granularity); for the latter it tracks the index register and extracts all offsets that taint those indexes. As DTA is not a lightweight technique, VUzzer uses it only at initialization on the seeds corpus and on inputs that exercise previously unseen basic blocks during the main fuzzing loop. DTA is at the core of VUzzer’s *magic byte detection* algorithm: at initialization it exploits the available valid inputs to extract bytes within the input placed at a certain (unique) offset in all valid inputs and later builds on this knowledge.

Ultimately, the knowledge acquired through static and dynamic analysis is exploited by recombination and mutation operators. VUzzer always applies a *single-point crossover* operator that, given two inputs,

breaks them at a randomly picked cut point and then recombines parts of different parents to produce two children. Mutation operators follows with a fixed probability, producing a new child from a single parent. Those operators are application aware, making use of results from static (e. g. immediate values of `cmp` instructions) and dynamic (e. g. magic bytes, offsets that taint the index operand of `lea` instructions) analyses.

2.3 SYMBOLIC-ASSISTED FUZZING

Automated test generation has been done since the 1970s with the aid of symbolic execution. Initial work focused on finding inputs to exercise a specific execution path within the SUT. More recently, symbolic execution has seen a resurgence in the context of security testing, given that it has been proven effective in exploring all feasible execution paths within real-world programs.

Algorithm 4: Symbolic-Assisted Fuzzing

Input: Initial seed t_s
Output: Set of crashing inputs C

```

1  $C \leftarrow \emptyset$ 
2  $Q \leftarrow \{t_s\}$ 
3 if RunAndCheck( $t_s$ ) then
4   |  $C \leftarrow C \cup \{t_s\}$ 
5 end
6 while  $Q$  is not empty do
7   |  $t \leftarrow \text{SelectNext}(Q)$ 
8   |  $G \leftarrow \text{ExpandExecution}(t)$ 
9   | foreach  $t_G \in G$  do
10  |   | if RunAndCheck( $t_G$ ) then
11  |   |   |  $C \leftarrow C \cup \{t_G\}$ 
12  |   |   end
13  |   | Score( $t_G$ )
14  |   |  $Q \leftarrow Q \cup \{t_G\}$ 
15  |   end
16 end

```

An abstract view of a generic symbolic-assisted fuzzing engine is given in [Algorithm 4](#). It maintains a set of inputs, initialized with the initial seed input (line 2), from which new inputs are generated. On

line 3 the function `RunAndCheck` runs the SUT with the given input and returns a true value if the program ended with unexpected results. If this is the case, the input that caused the failure is added to a set which is returned at the end of the procedure. Next, a loop operates until the set of working inputs is empty. First it selects an element from the set (through `SelectNext` on line 7) and then generating a set of inputs from the selected input on line 8. `ExpandExecution` symbolically executes the SUT with the supplied input and collects the path constraint. The constraint that identifies the i th branch in the execution is negated so that when the updated path constraint is solved for the symbolic variables, the new input is going to exercise the same path as its parent up to the $(i - 1)$ th branch and take the alternative branch on the i th one. Later, each generated input is run and checked for exceptions, before being scored and added to the set of working inputs (line 10 to 14). `Score` evaluates the given input and assign a score that can be used in successive calls to `SelectNext`.

The specifics of each component of [Algorithm 4](#) is different from one implementation to another. The most peculiar ones, that differentiate the most symbolic-assisted fuzzers among each other, are `SelectNext` (line 7), `ExpandExecution` (line 8) and `Score` (line 13). [DART \[51\]](#) provides a reference implementation for white box fuzzing. The set of working inputs consists at most of a single element (the seed input is randomly generated), `SelectNext` always returns the only element present and `ExpandExecution` returns a singleton set which input is the result of running the constraint solver on a modified version of the path constraint exercised by its given input; [Godefroid, Klarlund, and Sen](#), for exposition purposes, present a depth-first search on the execution tree of the SUT (i.e. the last constraint on the path constraint is negated before being fed to the solver), although other search strategies can be used (e.g. breadth-first, random or heuristics-based). When the program is symbolically executed with a generated input, [DART](#) checks that there are no divergences (i.e. the expected path is effectively taken) and if so generates a new input; otherwise it flips some flags tracking search incompleteness and restarts the procedure with a new randomly generated input.

SAGE [52, 53] expands upon the work on DART in a threefold manner: (i) by leveraging concolic execution on x86 binaries (in contrast to C source code); (ii) implementing a generational search in order to optimize the time spent doing symbolic execution; (iii) testing the SUT in its entirety instead of units of code. The set of working inputs is initialized with a well-formed, valid input (instead of randomly generated). `SelectNext` selects the input with the highest score. SAGE scores each input with the number of newly discovered basic blocks (e.g. input i is assigned a score of n if it exercises n previously unexercised basic blocks). `ExpandExecution` implements the generational search: for each constraint on the path constraint it generates a new input by negating it; to prevent redundancy in the sub-searches, a bound parameter for each sub-search is used to limit backtracking above the branch where the sub-search forked. The rationale behind this search strategy is to maximize the number of generated inputs from each symbolic execution (which are expensive for large programs with large amounts of symbolic variables); the authors report a mean time to complete a single symbolic execution run for one of the analyzed SUT of 25 minutes and 30 seconds while testing the same program takes seconds. Using a generational search strategy, SAGE is able to spend only 25% of the search time doing symbolic execution. Because of the nature of this search strategy, it is critical to the effectiveness of SAGE that the seed input exercises a path as deep as possible so that the first symbolic execution generates as much new inputs as possible.

The symbolic execution engine of SAGE is trace-based: the SUT is executed concretely and a trace of the execution is stored into a file; later this trace is interpreted symbolically and reasoned about. Other symbolic-assisted fuzzers (e.g. EXE [33] and KLEE [32]) take a different approach: the SUT is (usually) translated to an intermediate language and interpreted; at branching statements the state of the SUT is stored and execution is forked. These approaches are named respectively offline and online symbolic execution in literature. A disadvantage of the former is the redundancy in re-executing instructions symbolically, while the latter puts a strain on memory because

of the continuous forking of state. MAYHEM [36] takes instead an hybrid approach: the system starts with an online exploration phase in which normal online symbolic execution is performed; then, whenever memory utilization reaches a threshold, a checkpoint manager selects an active execution and stores a checkpoint containing only the symbolic execution state, freeing memory. When there are no more active executions, a checkpoint restoration phase selects a checkpoint and restores it in memory by re-executing the SUT only concretely, as the symbolic state was stored in the checkpoint. After this, the system restarts online exploration. This hybrid approach allows MAYHEM to be resilient to memory requirements while avoiding redundant and expensive symbolic executions, taking the best of online and offline symbolic execution.

EXE uses a coverage-based strategy to implement `SelectNext`: it selects the input that exercises a path that is blocked (i. e. waiting to be scheduled for execution) on an instruction that has been executed the fewest number of times; then the selected input and its children are expanded (i. e. `ExpandExecution`) in a depth-first manner. KLEE uses two different search strategies in a round robin fashion. Random path selection uses a random walk from the root of a binary tree representing all visited execution states and stops when it reaches a leaf (i. e. leaves are active states while internal nodes are places where execution forked). This strategy favors states high up in the tree, which have the favorable property of having less constraints on their symbolic inputs; moreover it makes it hard for the search to become stuck on states generated by a tight loop containing symbolic variables (what the authors call “fork bombing”). Coverage-optimized search uses heuristics to assign weights to states and uses them to perform random selection.

2.4 HYBRID TECHNIQUES

Hybrid techniques merge symbolic-assisted fuzzing with black box mutational fuzzing (i. e. random testing). Hybrid Fuzz Testing [89] uses an initial round of symbolic execution to find its way to a fixed,

configurable, number of “frontier nodes”; for each node a path constraint is collected (i. e. a frontier node is the basic block at the end of one of those paths). Then new inputs are randomly generated that still respect the path predicates on each frontier node; these inputs are then executed concretely to check for exceptions. While this approach can help exploring the execution tree in its breadth early on (something random testing struggles to achieve), random testing can get stuck on deeper checks that have not been explored by an initial symbolic execution run.

The system described in “Hybrid concolic testing” [81] alternates random testing with symbolic execution. It starts with random testing and whenever no new coverage is produced in a predefined number of steps, it switches to symbolic execution from the current program state. Both random testing and symbolic execution are implemented in an online manner; random concrete values, as well as symbolic values, are fed as input to the SUT without re-executing instructions to build the symbolic state before switching. Hybrid concolic testing implementation is based on the online concolic executor CUTE [97].

More recently, Stephens et al. presented Driller [101], which composes a CGF with a symbolic executor by using state-of-the-art implementations and optimizations to expand over the intuition of Majumdar and Sen in “Hybrid concolic testing.” Driller uses AFL as CGF and angr [98] as symbolic executor. As with hybrid concolic testing, Driller starts off with AFL; whenever AFL communicates that it has no pending favorites inputs waiting in the queue (i. e. when the property `pending_favs` in AFL statistics file reaches zero), Driller invokes the symbolic executor. This in turn traces the execution of the unique inputs found by AFL and tries to explore its neighborhood in the execution tree before returning newly generated inputs to AFL. To motivate their approach, the authors define a “compartment” within a program as a portion of code locked behind strict checks (e. g. an if statement checking for a magic byte). Fuzzing is powerful at effectively exploring portions of code *within* a compartment but struggles to pass those checks that would allow it to explore deeper compartments. Symbolic execution on the other hand complements fuzzing

as checks that separate compartments are easily solved in a single symbolic pass but exploring inside a compartment takes as much resources. Driller uses fuzzing to cost-effectively explore portions of code within compartments and selective symbolic execution to switch compartments.

Another key optimization of Driller over hybrid concolic testing is given by the symbolic exploration phase. Instead of switching back to fuzzing as soon as symbolic execution finds a new input, Driller lets symbolic execution run until a configurable number of new basic blocks has been discovered. The authors observed that in some cases Driller was able to generate inputs that get only partially through checks that are code in succession, causing the fuzzer to rapidly get stuck again on a new check; because symbolic execution is expensive, this frequent switching caused Driller to perform poorly. The symbolic exploration phase alleviates this by allowing symbolic execution to work through multi-checks in one go.

2.5 COOPERATIVE FUZZING

With the term *cooperative fuzzing* we refer to a system that runs in parallel a number of possibly different fuzzer instances able to share information with each other in order to achieve a common goal (i. e. increase global code coverage). The intuition behind this is threefold: (i) due to the non-deterministic nature of fuzzing, results across different runs of the same fuzzer may vary greatly; (ii) different fuzzers may yield different results for different programs (i. e. there is no fuzzer that works best for all possible programs); (iii) providing a mean to share information may enable a single fuzzer to obtain critical knowledge that otherwise would not have obtained or would have obtained much later. By parallelizing fuzzers execution and enabling an overlay of communication features, cooperative fuzzing tries to exploit non-determinism and implementation differences among fuzzers with the objective of creating an higher-level fuzzer.

An example in literature of such system is presented in [24, 25]. By making an analogy between foraging behaviours in biology and

bug hunting in software, the authors propose a system that models fuzzing as Lévy flights over the input space of the SUT; a Lévy flight is a widely used mathematical model which, under some hypotheses, minimizes search times when applied to resources foraging. A Lévy flight is formally defined as a Markovian process having independent and stationary increments; intuitively it describes the movement of a particle that at each step randomly chooses direction and length of the next step to make. Assuming a bit-string of length N , split into n segments of length $\frac{N}{n}$, each fuzzing instance simulates two Lévy flights: one over the domain of the segment offsets $\{1, \dots, n\}$ and one over the domain of segment values $\{1, \dots, 2^{\frac{N}{n}}\}$. For each of these, the distribution of flight lengths is given by the power law:

$$p_i(l) \sim |l|^{-1-\alpha_i} \quad i = 1, 2$$

where $0 < \alpha_i < 2$ is a parameter that controls the diffusivity of the stochastic process. For lower values of α_i the process exhibits bigger steps in the search space, while for higher values the distribution of step lengths moves its probability mass toward zero. During fuzzing, these parameters are self-adapted so that if the fuzzer is exploring an high-quality region of space, it should make smaller steps in order to thoroughly explore the current region (i. e. sub-diffusion); on the other hand, if the fuzzer is exploring a low-quality region, it should make bigger steps to try to explore different regions in the hope of finding an higher-quality one (i. e. super-diffusion). The quality metric used is the number of basic block exercised by the given input, that were not exercised by any previously tested input.

At each iteration, a fuzzer instance generates k_{gen} new inputs by simulating a step in offset space and one in segment space and replacing the segment at the given offset with the given segment value in a given input. The input that is modified is seeded on the first iteration and the latest generated input successively. These new inputs are then evaluated against a queue of inputs the fuzzer maintains using

the basic block metric. Diffusivity parameters are then updated using the following:

$$\alpha_i = \frac{2}{1 + e^{b_i - E(X_{\text{gen}}, X_{\text{all}})}} \quad i = 1, 2$$

where $b_i \in \mathbb{R}^+$ is a fixed parameter that relates the quality measure to diffusion behaviours (i. e. at which quality value the search switches from sub- to super-diffusion or vice versa) and $E(X_{\text{gen}}, X_{\text{all}})$ is the newly generated inputs quality, measured against a queue of k_{max} older inputs. Then the queue X_{all} is updated with inputs from X_{gen} in a first-in-first-out manner, keeping a total of k_{max} items.

A swarm of fuzzers so described runs in parallel; after all fuzzers complete one iteration, a k-means clustering algorithm finds k clusters of fuzzers by using the latest generated input of each. For each cluster, all fuzzers are relocated to the same position of the fuzzer with the best quality evaluation in the cluster (i. e. resets the starting position of Lévy flights with the latest steps of the best neighbour).

Another cooperative fuzzing system by the same author, Böttinger, is presented in [23, 26]. Once again the system is inspired by biology: chemotaxis is the movement of organisms of a colony attracted by a chemical trace; two kinds of fuzzers are deployed in a cluster (i. e. they run in parallel): explorers (i. e. CGFs) use coverage feedback to produce a trace which is followed by workers (i. e. black box mutational fuzzers). The feedback-enabled explorers, having more information at their disposal, are more easily able to explore new portions of code. Information of their findings is encoded in a trace which workers are attracted to, effectively guiding the faster black box fuzzing instances toward new regions. The intuition behind this approach is reminiscent of Driller's, where fuzzers of different natures are used to cope with each other's limitations.

COOPERATIVE FUZZING FRAMEWORK

In this chapter we describe the design of our Cooperative Fuzzing Framework (CFF) and details about its implementation. We are going to show how instances of different fuzzers can communicate with each other in order to share knowledge and cooperate to improve the overall fuzzing efficacy (in terms of coverage and number of unique bugs found).

3.1 SYSTEM DESIGN

The CFF is designed to run fuzzers of different natures in parallel, harvest knowledge from each instance and intelligently broadcast it to selected other instances. This framework allows construction of algorithms that orchestrate the information flow among the fuzzing instances. We design two primitives that allow a fuzzer to share knowledge with other fuzzers and devise a distributed architecture that employs these primitives to manage the information flow among fuzzer instances in a smart manner. The architecture is modular so that inner components can be replaced as needed, moreover it can be deployed across multiple machines.

3.1.1 *Common Fuzzer Interface*

We require fuzzers to implement two primitives that allow our framework to interface with them. Having a common interface enables the framework to operate with different fuzzers, as long as they adhere to the Application Programming Interface (API).

All fuzzers work with test-cases and CGFs keep a queue of test-cases adding elements to it if they (loosely speaking) increase coverage. Whenever a fuzzer finds a test-case that expresses some in-

interesting property, stores it into the internal queue. This property is deemed interesting in a subjective manner, based on how the fuzzer operates and the knowledge it has already acquired about the SUT. It follows that we can track the progress of a fuzzer by intercepting and analysing interesting test-cases; in other words we can **extract** knowledge from the fuzzer. The first primitive that we require a fuzzer to expose to the API enables exactly this: we require a fuzzer to store interesting test-cases to the file system. This choice was made since most CGFs already fulfill this requirement; this is because the resulting corpus can be reused for future fuzzing campaigns against the same software or another that accepts the same input format without starting from scratch.

The second primitive we require a fuzzer to implement, allows our cooperative framework to **inject** test-cases into it. This mechanic is inspired by that of pollination, where pollen carrying the genetic material of one plant is transferred to another plant to be later fertilized. The injection primitive in our framework allows one test-case, carrying some knowledge about the fuzzer instance it comes from, to be merged with the knowledge of the receiving fuzzer instance. As with the extraction primitive, injection operates through the file system: the fuzzer needs to periodically check a predetermined folder and import new test-cases into memory.

Different fuzzers may work at different paces: traditional CGFs *tick* after each new test-case is generated; fuzzers that follow the evolutionary approach advance a time-step when a new generation gets synthesized (e. g. after several test-cases are generated). In between successive time-steps, a fuzzer is not able to accept and process external test-cases. Thus there is a need for a fuzzer to communicate through the cooperative framework when it is able to accept new input (e. g. when the inject primitive can be called on this fuzzer instance). We assume that a fuzzer is always able to accept new test-cases, otherwise it needs to implement a third interface endpoint: when the fuzzer can accept new input, it will broadcast this information to all other connected fuzzers. This information allows for a form of **congestion control** within our cooperative framework and is

entirely delegated to the fuzzer in order to abstract over its implementation details.

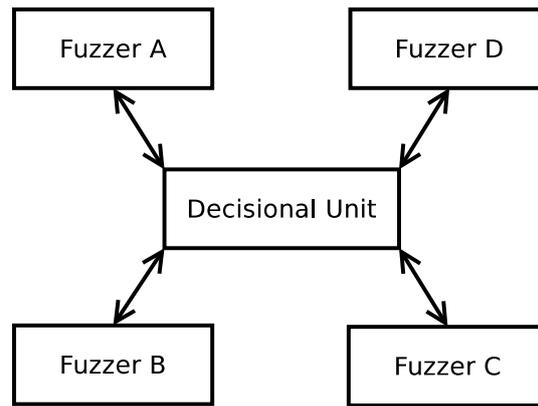
With these two primitives in place (not counting the latest introduced, which is optional), our cooperative framework is capable of interacting with the running fuzzer instances by directly accessing an uniform communication layer that uses test-cases as an atomic piece of information. [Table 3.1](#) summarizes the CFF interface with fuzzer instances.

Primitive	Description	Initiated by	Required
Extract	Extracts interesting test-cases from fuzzer	Fuzzer	Yes
Inject	Injects test-cases into fuzzer	Framework	Yes
Congestion Control	Signals to the framework when the fuzzer is able to consume injected test-cases	Fuzzer	No

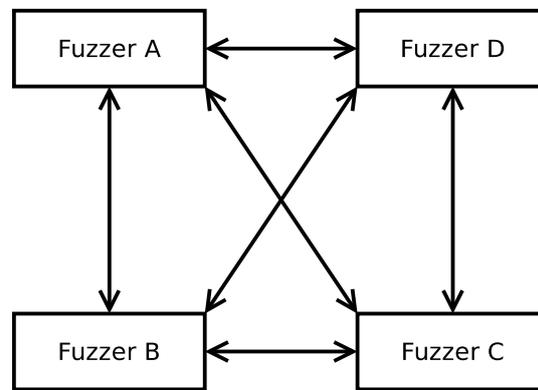
Table 3.1: Cooperative Fuzzing Framework interface with fuzzers

3.1.2 Central Decisional Unit

With a common interface with fuzzers in place, the next component we need to devise is a decisional unit responsible of controlling the information flow among fuzzer instances. This component acts as an intermediary or a filter for the interface between two fuzzers, applying a pre-determined strategy that decides which test-cases extracted from a fuzzer should be injected into which fuzzer (if any). As some fuzzers need some form of congestion control, the decisional unit is also responsible of deferring the call to the injection primitive. As an example, when an evolutionary CGF is in the set of fuzzer instances, one strategy might choose to keep track of the best (according to some metric) test-case extracted from the other running fuzzers, to inject it only when the evolutionary fuzzer is able to consume injected test-cases; another strategy might choose to do the same only for a subset



(a) Physical view of the framework communication model. Running fuzzers communicate only with the central decisional unit which selectively broadcasts messages to other fuzzers



(b) Logical view of the framework communication model. Information can flow across all running fuzzers

Figure 3.1: Cooperative Fuzzing Framework communication model

of running fuzzers and so on. We note that for strategies such as those briefly described above, we need the decisional unit to be centralized: communication does not happen between fuzzer instances directly, but everything is coordinated by a **central decisional unit** that models communication in accordance to an user-defined strategy.

Figure 3.1 gives a graphical representation of the communication model in the CFF. More specifically, Figure 3.1a depicts the *physical* links between components of the framework; Figure 3.1b instead shows how, on the *logical* level, each running fuzzer can directly exchange test-cases with other running fuzzers. In this configuration each running fuzzer communicates directly only with the central decisional unit by means of the common interface described in

Section 3.1.1, without having any knowledge about other running fuzzers.

3.1.3 Cooperative Fuzzing Strategies

With the architectural design in place, we can proceed to describe a more detailed view on the CFF operation.

Algorithm 5: Generic strategy for the Cooperative Fuzzing Framework

Input : Set of running fuzzers F . Set of fuzzers that need congestion control F_c

```

1  foreach  $f \in F_c$  do
2  |    $W_f \leftarrow \emptyset$ 
3  end
4  while all fuzzers are running do
5  |   if new test-case  $t$  from a fuzzer  $f_t$  then
6  |   |    $S \leftarrow \emptyset$ 
7  |   |   foreach  $f \in F \setminus \{f_t\}$  do
8  |   |   |    $s \leftarrow \text{Score}(f, t)$ 
9  |   |   |   if  $f \in F_c$  then
10 |   |   |   |    $W_f \leftarrow W_f \cup \{(t, s)\}$ 
11 |   |   |   |   else
12 |   |   |   |   |    $S \leftarrow S \cup \{(f, s)\}$ 
13 |   |   |   |   end
14 |   |   |   end
15 |   |   foreach  $f \in \text{Winning}(S)$  do
16 |   |   |    $\text{Inject}(f, t)$ 
17 |   |   end
18 |   |   end
19 |   |   foreach  $f \in F_c$  do
20 |   |   |   if  $f$  is ready to receive inputs then
21 |   |   |   |   foreach  $t \in \text{WinningCongestion}(W_f)$  do
22 |   |   |   |   |    $\text{Inject}(f, t)$ 
23 |   |   |   |   |   end
24 |   |   |   |    $W_f \leftarrow \emptyset$ 
25 |   |   |   end
26 |   |   end
27 end

```

Algorithm 5 gives the generic template of a cooperative fuzzing strategy. There are three user-defined functions that describe which test-cases should be forwarded to which fuzzer (in other words they

define the strategy). When a fuzzer stores a new interesting test-case, the extract primitive gets executed and the cooperative framework will pick it up. Then, for each other running fuzzer, the Score function will assign a numeric value representative of some measurement on the test-case in relation to the fuzzer's state. These numeric values are then used in two *winner selection* procedures: *Winning* selects a set of fuzzers that are not under congestion control into which the newly extracted test-case is going to be injected; *WinningCongestion* instead, for each fuzzer under congestion control which is ready to receive new inputs, selects a set of test-cases within the window set W_f of inputs collected since the previous round.

The Score function of [Algorithm 5](#) is a core component around which the design process of a cooperative strategy rotates. When devising a new strategy, one would start by defining a scoring function as the winner selection procedures will use its values to make the final decision. The scoring function is fundamental from another point of view: it has been designed to be analogous to the fitness function found in EAs, where in our case *fitness* represents how much would a fuzzer benefit from receiving a given test-case.

As for CGFs, the CFF uses code coverage as a basis for the scoring function. A strategy might choose to keep track of all basic blocks discovered by each running fuzzer (this is possible because *interesting* test-cases often execute previously unseen basic blocks); the scoring function can then return a numeric value representing a specific property of the newly extracted test-case in relation to the fuzzer's current knowledge (the set of discovered basic blocks). Behind the intuition that seeding a fuzzer with a test-case that exhibits new basic blocks might help the fuzzer discovering even more new basic blocks, we might construct a strategy to broadcast test-cases to those fuzzers for which there is at least one undiscovered basic block. In this context, we can set the scoring function to return the number of basic blocks new to the fuzzer and select as winners fuzzers with a positive score.

The *Winning* function of [Algorithm 5](#) is responsible for working out a set of running fuzzers, given the results of the scoring function. Fuzzers from this set, which can be empty, are to be injected with the

test-case. This function resembles the selection stage of EAs: selects fuzzers (in contrast to selecting individuals of a population) which are going to receive a new test-case, so that the fuzzer itself can apply its own mutation algorithms to try to generate new interesting test-cases. The `WinningCongestion` function on the other end, instead of working with a set of scores for different fuzzers over a single test-case, selects a set of scored test-cases to be injected into a single fuzzer.

3.2 SYSTEM IMPLEMENTATION

This section describes our implementation of the CFF, which is composed of two executables: a *driver* and a *master*. A driver is responsible to run, monitor and interact with a fuzzer instance (i. e. via the API defined in [Section 3.1.1](#)) while the master implements the central decisional unit as defined in [Section 3.1.2](#); together, drivers and master, implement the CFF strategy described in [Section 3.1.3](#). The full implementation, which is available on Github¹, consists of a total of 2,832 lines of Rust, C and Bash source code.

3.2.1 Communication Channels

Driver and master are implemented as two separate executables to fulfill the requirement of being able to deploy the different components across different machines. To support this, drivers communicate with the master using three separate distributed message queues as depicted in [Figure 3.2](#); these communication channels are implemented over ZeroMQ², a popular asynchronous messaging library with binding for a large variety of languages, including Rust and C. All messages over these channels are serialized as space-separated values. The interesting channel is a queue shared among all drivers on which interesting inputs captured by drivers are pushed to be later

¹ <https://github.com/acidghost/uberfuzz2>

² <http://zeromq.org/>

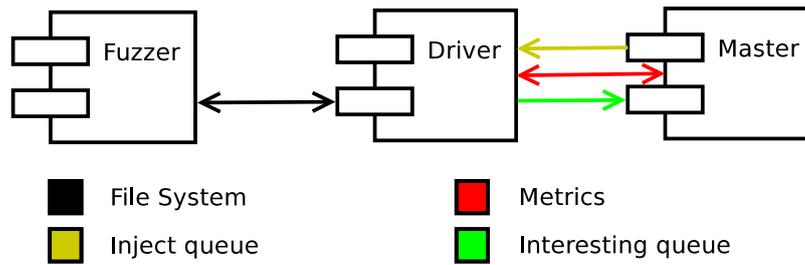


Figure 3.2: Communication channels between fuzzer, driver and master.

pulled by the master. Messages on this channel have the following fields:

`fuzzer_id` identifies the fuzzer instance that generated the input;
`input_path` the path to the input deemed interesting by the fuzzer;
`coverage_path` path to a binary file containing the coverage information generated by executing the SUT with the input at `input_path`.

The inject channel is, again, shared among all drivers and is set up so that master publishes messages to which drivers can subscribe. Messages on this channel represent inputs that specific fuzzers need to inject into their queue of working inputs. Each message is made up of the following fields:

`fuzzer_ids` a list of fuzzer identifiers, separated by `'_'`, that need to receive and inject the input represented by this message;
`input_path` path to the input to be injected;
`coverage_path` path to coverage data exercised by the given input.

Finally, the metrics channel uses the request-reply pattern: each driver binds to a unique port (i.e. the channel is not shared among drivers) in reply mode and the master connects to it in request mode. Whenever the master needs a metric evaluation from a specific fuzzer, it sends a metric request to the appropriate driver and synchronously waits for a reply. Both request and reply messages are comprised of a single field: requests contain a `coverage_path`, as described for

other message types; replies contain instead a floating point number, representing the metric evaluation result.

Communication between a fuzzer and its driver is achieved through the file system. This is because most fuzzers use the file system to store their data. More specifically “interesting” inputs, as well as inputs to be fuzzed, are written to a folder (which might be the same for both). Follows that to capture new interesting inputs, a driver needs to track new files added to the appropriate folder. On the other hand, to inject new inputs into the fuzzer’s queue, a driver simply needs to add a file with the input to the right folder. The Linux `inotify` subsystem is used to avoid doing inefficient long polling to capture fuzzer events (e.g. when a new interesting input is added). For fuzzers that need congestion control, we require them to write (i.e. drivers listen for `inotify IN_MODIFY` events) to a predetermined file whenever there is availability to receive new inputs.

3.2.2 *Driver Implementation*

Drivers are responsible to launch and interact with a fuzzer instance running on the same machine. The driver program is written in 1,496 lines of C code; Rust was the initial choice but unfortunately interfacing with Intel BTS via Foreign Function Interface (FFI) proved to be harder than needed: execution traces were empty and debugging such low-level kernel features is notoriously hard. As the only hard requirements were performance and a reliable interface to BTS, we decided to code the driver in C, which is notoriously fast and can directly interface with BTS via the `perf_event_open` Linux system call. The choice of using BTS over the more modern Intel PT is merely due to the unavailability of a more modern processor to conduct experiments with. Nonetheless, for our purpose, BTS offers the same features as PT does.

The driver starts up by initializing communication channels: connects to interesting and inject queues, binds to a port to serve metric replies and sets up `inotify` watchers. Next it runs the assigned fuzzer with the given configuration and starts the main loop, which

runs until the fuzzer terminates or an external interrupt or kill signal is received. The main loop first checks if there are new `notify` events and if so, for each new file, it executes the SUT with BTS enabled; this produces an execution trace in the form of a sequence of branches which is stored to a file. This new interesting input is then broadcast over the interesting queue, along with its coverage information.

Next, the driver checks if there is a new request on the metric channel and, if so, sends a reply. This is done by reading the coverage file referenced in the request and evaluating it with respect to the accumulated driver knowledge: whenever a new interesting input is processed, the driver updates an hash table that maps branches to an hit counter; this enables the driver to track already discovered branches and how many times each has been hit, across all intercepted interesting inputs. In the current implementation, the metric is given by the number of newly discovered branches.

The last step in the driver's main loop consists in trying to pull one message from the inject queue. If a message is successfully read and is addressed to itself (i. e. `fuzzer_ids` contains the right identifier), the driver adds the attached coverage information to the internal hash map and injects the input file into the fuzzer by creating a copy of the file in the fuzzer's working directory. Most fuzzers already implement some kind of file system synchronization to import test cases: AFL and its extensions have a mechanism, used to synchronize instances running in parallel, that periodically checks a directory for new files and imports them; VUzzer writes all test cases after each generation and loads them again at the beginning of the next one. Honggfuzz, which is also used in the evaluation of CFF in [Chapter 4](#), does not implement any such mechanism: we use a forked version of Honggfuzz which implements a basic synchronization component similar to AFL's.

As a final remark, the driver has no knowledge of the kind of fuzzer it is interfacing with, instead it exposes a collection of command line parameters flexible enough to accommodate a variety of fuzzers. These parameters are filled in by the master when it executes a driver. The name "driver" is inspired by device drivers which

abstract away from the OS the details of interfacing with different hardware; here a driver abstracts away implementation details of interfacing with a *generic* fuzzer, minus some configuration parameters. Also, these configuration parameters are not encoded in the master either but in external configuration files.

3.2.3 *Master Implementation*

In our implementation of the CFF, the master is the main executable; responsible of running and interacting with the drivers, consists of 749 lines of Rust source code.

The master starts up by binding to the interesting and inject queues before starting the drivers, parameterized for the specific fuzzer by a mixture of command line arguments and configuration files. Moreover, it sets up `inotify` watchers for fuzzers that need congestion control. The fact that this direct communication channel between master and fuzzer exists may hinder the applicability of the current implementation in a truly distributed environment; nonetheless, the limitation can be easily circumvented by employing some kind of middleware or update the CFF implementation by introducing a new ZeroMQ communication channel between driver and master.

The main loop executes until an external interrupt is received or one of the driver terminates early. At each iteration of the loop a new item is pulled from the interesting queue. The pulled input is then evaluated against all other fuzzers by sending a metric request to all drivers excepts the one that sent the interesting input and synchronously waiting for a reply. When all evaluations have been collected a competition to win the input starts. Two generic strategies are currently implemented: one that selects the driver that replied with the highest or the lowest metric; one selects drivers if their evaluation is greater or less than a threshold. The winning strategy, along with threshold and ordering parameters, are configurable through command line arguments. Once winning drivers are selected, the master sends a new message on the inject queue with the `fuzzer_ids` field set to the winning drivers' `fuzzer_id`.

To implement the congestion control mechanism, the master keeps an hash table mapping drivers that need this feature to a list of interesting inputs and their evaluation by the mapped driver. When the master receives the signal through `inotify`, it starts a competition to select an input from the aforementioned list and sends it to the driver before clearing the list.

EVALUATION

This chapter presents an evaluation of four different fuzzers and our implementation of the Cooperative Fuzzing Framework (CFF) as described in [Section 3.2](#); the implementation uses the same four fuzzers in order to draw meaningful comparison results. In [Section 4.1](#) we compare the results, in terms of coverage, of running fuzzers without cooperation; [Section 4.2](#) presents results of running the same fuzzers with cooperation. Lastly [Section 4.3](#) presents an evaluation of cooperative fuzzing in terms of crashes found.

CHOICE OF FUZZERS For the choice of fuzzers we decided to use AFLFast, FairFuzz, Honggfuzz and VUzzer. The first two provide two different implementations of the state-of-the-art CGF, AFL; we argue that two different implementation of the same core algorithm may yield different results. Honggfuzz is chosen as it provides different means to extract feedback from the SUT, possibly resulting in a different feedback signal (given the same input) compared to other methods (e. g. AFL uses QEMU). Lastly, VUzzer provides with an evolutionary fuzzer which uses static and lightweight program analysis; because of this, CFF uses the congestion control mechanism to interact with VUzzer. All fuzzers were run with the default parameters and configuration, with the exception of those required to correctly interface with the SUT. Moreover, as already mentioned in [Section 3.2.2](#), all these fuzzers (besides Honggfuzz) support a file system based synchronization mechanism to periodically import new test cases; for Honggfuzz, we use an extended version that mimics AFL's synchronization component.

EXPERIMENTAL INFRASTRUCTURE Experiments were run on a 64-bit machine with 8 cores (running at 3.4GHz), with 16GB of RAM,

running Ubuntu 16.04. Because of a limitation of VUzzer implementation, we had to run it inside a virtual machine hosting Ubuntu 14.04. To allow communication with its driver running on the host machine, we used a shared folder; this allowed us to reuse the inotify infrastructure as if the fuzzer was running locally. Finally, we disable Address Space Layout Randomization (ASLR) before fuzzing and when running the crash triaging script; in the latter, we also limit the available address space to simulate the same environment of the fuzzer that found the crash.

TESTING TARGETS We chose a set of programs that are widely used both in practice and literature:

`djpeg` uses the popular `libjpeg-turbo` (version 1.5.1) and has been used for the evaluation of FairFuzz and VUzzer;

`objdump` is a component of the suite `binutils` (version 2.28) which has also been tested by AFLFast and FairFuzz; we test it with the command line option `-d`, which provides a disassembler;

`tiff2pdf` from the popular `libtiff` (version 4.0.9), parses a TIFF image and converts it to a PDF; we use it without command line arguments;

`listswf` from the popular `libming` (version 0.4.8), parses a file in SWF format; we use it without command line arguments.

Moreover, as VUzzer requires a minimum set of seed inputs to work properly, we randomly chose the minimum required by VUzzer from test samples provided with each SUT; these inputs were used to also seed other fuzzers.

As an additional information, we report on the number of basic block and functions found by static analysis via Radare¹ in [Table 4.1](#).

¹ <https://rada.re/>

SUT	basic blocks	functions
djpeg	6187	366
objdump	43211	2220
tiff2pdf	11129	791
listswf	3349	446

Table 4.1: Number of basic blocks and functions for the chosen targets.

4.1 SINGLE FUZZER EVALUATION

In this section we present an evaluation of the chosen fuzzers, without any cooperation. We ran each fuzzer independently for 24 hours on each of the targets, except for `listswf` which ran for 6 hours. [Table 4.2](#) reports the arithmetic mean and the 95% confidence intervals for the number of unique basic block transitions as computed by Intel BTS over five rounds. The coverage values are aggregated over time intervals of one minute.

SUT	AFLFast	FairFuzz	Honggfuzz	VUzzer
djpeg	3739.4 ± 113.831	4043.2 ± 103.838	4112.8 ± 39.5483	2801 ± 53.5514
objdump	4762.6 ± 23.3693	5067 ± 62.6832	4132.4 ± 104.8	3162.2 ± 138.462
tiff2pdf	8971.2 ± 152.865	8813.8 ± 146.756	5260.2 ± 148.591	3616 ± 34.6427
listswf	6831.6 ± 2615.24	8586.8 ± 87.7467	6345.6 ± 2358.52	5048.2 ± 90.3928

Table 4.2: Mean coverage with 95% confidence intervals for single fuzzers. Highlighted is the best for the given program.

By looking at the table, as well as at [Figure 4.1](#), which presents the evolution of coverage over time, it is easy to see how no fuzzer is decisively better than the others across all tested programs. This validates, although for a small sample of programs and fuzzers, our intuition based on the no free lunch theorem, for which there is no fuzzer that performs better than all others across all possible test programs.

To derive more robust and meaningful insights over these results, we employ the Bayesian estimation model proposed in [73]. This model provides, among others, estimates of the posterior distributions of the means and their differences of two given sets of observations. [Figure 4.2](#) shows the distribution of the difference of means of Honggfuzz against AFLFast and FairFuzz for `djpeg`. The figure also

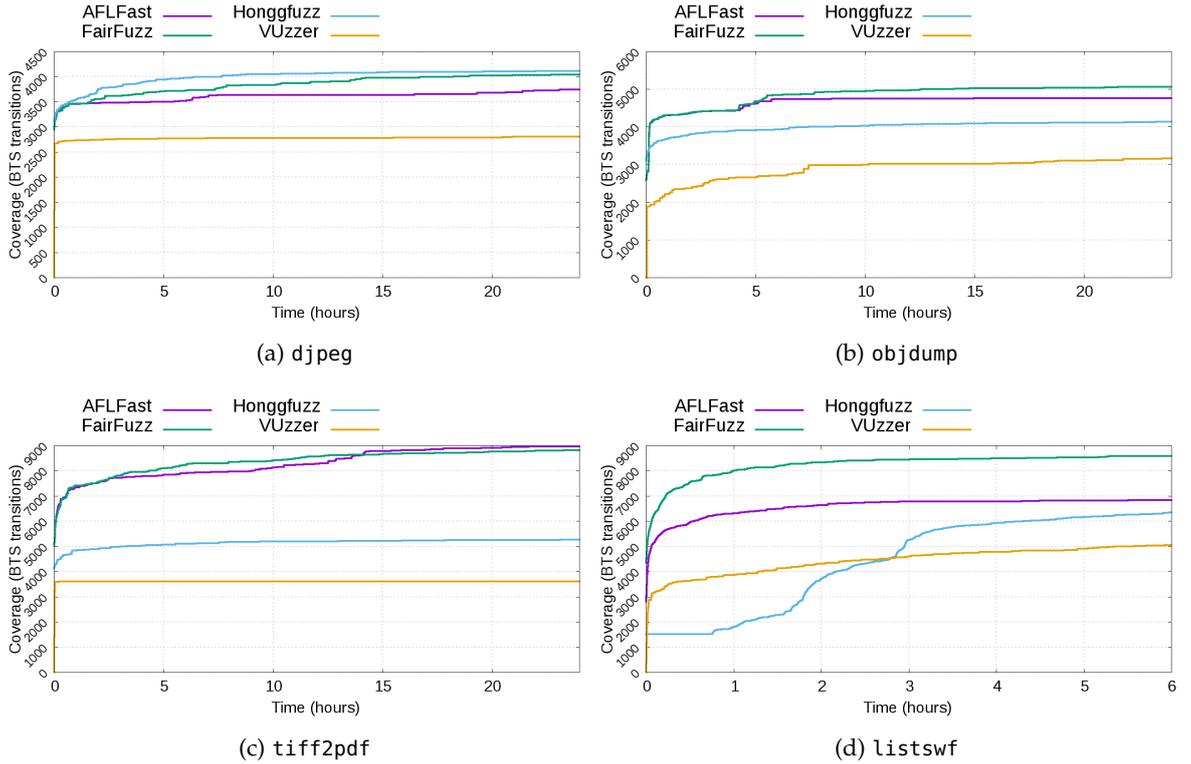


Figure 4.1: Mean coverage over time for single fuzzers.

shows the 95% Highest Density Interval (HDI), which represents the interval of values onto which 95% of the probability mass lies. [Figure 4.2a](#) clearly shows that the HDI lies completely above zero, meaning that with high credibility we can say that Honggfuzz performs better than AFLFast over djpeg. Unfortunately the same conclusion cannot be made for the comparison of Honggfuzz against FairFuzz. [Figure 4.2b](#) shows that a difference of means of zero lies on the HDI; moreover an estimated 20.2% of the probability mass lies below zero (i. e. in favor of FairFuzz).

[Figure 4.3](#) shows the difference of means for objdump. For both the cases of FairFuzz against AFLFast ([Figure 4.3a](#)) and FairFuzz against Honggfuzz ([Figure 4.3b](#)) the results strongly support the claim of FairFuzz uncovering more basic block transitions for objdump.

[Figure 4.4](#) shows the difference of means for tiff2pdf. Unfortunately, as for djpeg, we are not able to decisively confirm whether AFLFast is better than FairFuzz ([Figure 4.4a](#)). We can be instead more

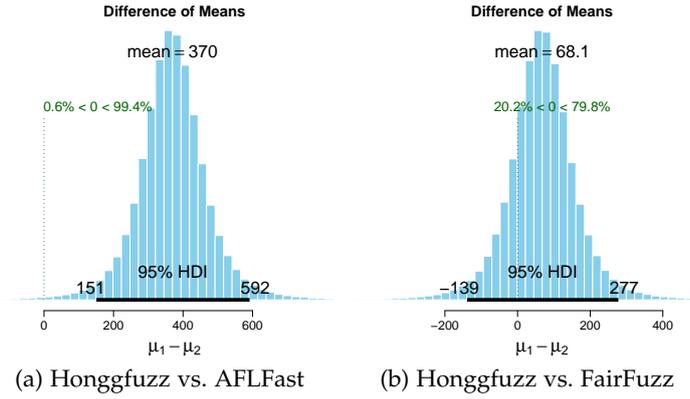


Figure 4.2: Single fuzzers: distribution of difference of means for djpeg.

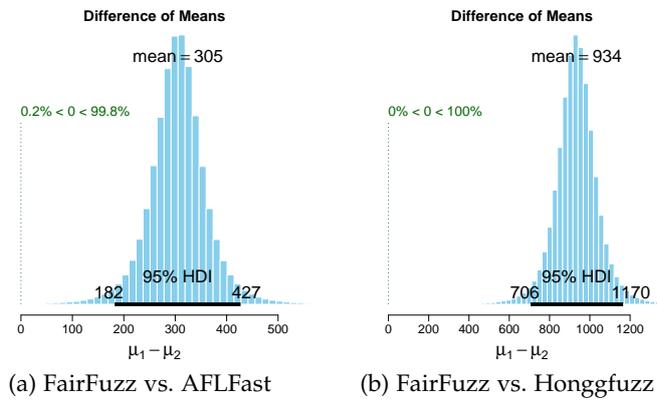


Figure 4.3: Single fuzzers: distribution of difference of means for objdump.

certain affirming that AFLFast outperforms Honggfuzz for the given SUT (Figure 4.4b).

To validate differences among fuzzers even further, we compare, in Table 4.3, the best single fuzzer with the result of taking the union of coverage traces across all four fuzzers for each time step. The table clearly shows that fuzzers uncover unique basic block transitions that are not exposed by any other fuzzer (i. e. uncover disjoint sets of transitions) and this contributes to reaching an higher final coverage consistently across all examined programs.

To delve deeper into these values, we show the result of Bayesian estimation in Figure 4.5. The 95% HDI falls above zero (i. e. in favour of the union of fuzzers) for all considered programs except djpeg, for which the results are inconclusive.

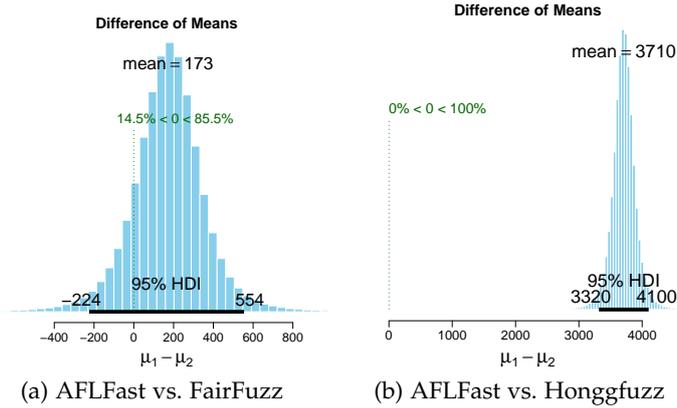


Figure 4.4: Single fuzzers: distribution of difference of means for tiff2pdf.

SUT	best single	union
djpeg	4112.8 ± 39.5476	Honggfuzz 4157.2 ± 40.0495
objdump	5067 ± 62.6821	FairFuzz 5404.6 ± 38.1997
tiff2pdf	8971.2 ± 152.8626	AFLFast 9695 ± 129.7239
listswf	8586.8 ± 87.7451	FairFuzz 8916.6 ± 83.8365

Table 4.3: Mean coverage with 95% confidence intervals for best single fuzzer and union of coverage traces.

4.2 COOPERATIVE FUZZING EVALUATION

In this section we present an evaluation of the efficacy of cooperation. For this purpose we compare the results of running the CFF for 6 hours with the union of coverage from the four fuzzers running without cooperation, as we have done in [Section 4.1](#). To make the comparison fair, for the union we take the maximum coverage after 6 hours. We configured the master to execute two distinct winning strategies: one selects the single fuzzer with the highest metric reply; the other selects all fuzzers for which the metric reply is greater than zero. In case of a draw, the pool of candidate winners is randomly shuffled and the first one is selected. Recall from [Section 3.2.2](#) that the metric is given by the number of undiscovered branches exercised by the given input (the metric is the same for all drivers). In this section we compare the results of running both strategies.

[Table 4.4](#) presents the final mean coverage with 95% confidence intervals. Looking only at the final mean we can see that one of the

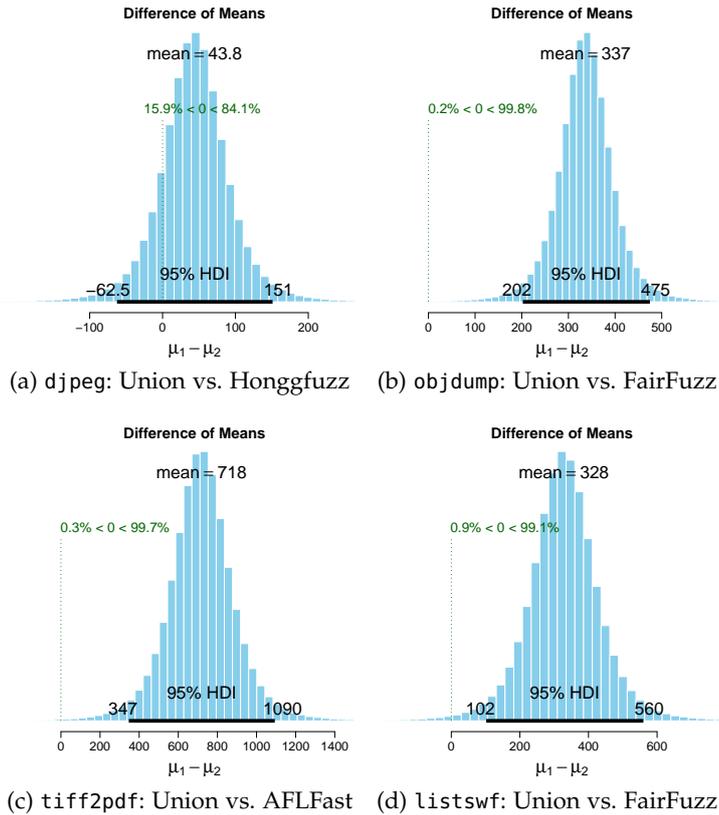


Figure 4.5: Distribution of difference of means for union of fuzzers against the best single fuzzer.

two cooperative strategies outperforms the union of fuzzers for all programs, but none of the two outperforms the other. Moreover for djpeg and objdump the union of fuzzers seems to perform the worst while for tiff2pdf and listswf it is the single winner cooperative strategy to be outperformed. For djpeg we see that the differences are too small to be sensitive enough as is confirmed by Bayesian estimation in [Figure A.1](#) and [Figure A.2](#). By observing the mean coverage over time, shown in [Figure 4.6a](#), we see that the union of fuzzers struggles against both cooperative strategies for the first two hours, before catching up.

The mean final coverage for cooperative strategies may seem to be better than the union of fuzzers for objdump ([Figure 4.6b](#)) as both means are higher and their confidence interval do not overlap with the one for the union. Unfortunately Bayesian estimation does not support these conclusions with high credibility (i. e. the HDI for the difference of means includes zero) as shown in [Figure A.3](#) for the

SUT	multi	single	union
djpeg	4056.4 ± 76.9499	4078.4 ± 85.6738	4028.6 ± 47.7396
objdump	5414.6 ± 224.121	5529.6 ± 338.651	5035.6 ± 54.5944
tiff2pdf	8765.6 ± 183.682	8577.6 ± 99.2457	8623.2 ± 183.399
listswf	9008.4 ± 122.81	8801.4 ± 96.4045	8916.6 ± 83.8381

Table 4.4: Mean coverage with 95% confidence intervals for winning strategies that select single or multiple winners and without cooperation.

single winner strategy against the union of fuzzers and [Figure A.4](#) for the multiple winners strategy against the union. In both cases, however, respectively 95.4% and 96.7% of the credible values of the difference of means are greater than zero (i. e. in favour of the cooperative strategy).

In the case of `tiff2pdf` and `listswf` we are presented with similar results as in both cases the single winner strategy is outperformed by the other cooperative strategy and the union of fuzzers. [Figure A.6](#) and [Figure A.7](#) show the results of Bayesian estimation on `tiff2pdf` for the multiple winners strategy against the union and the single winner strategy respectively; either case provides with means to say with an high degree of certainty that the multiple winners strategy performs better than the other. The coverage over time presented in [Figure 4.6c](#) suggests one more thing: as time progresses, the distance between the multiple winners strategy and the other two seems to increase; an evaluation over a period of time longer than 6 hours might reduce the variability of final coverage and yield more conclusive results. Moreover we note how none of the curves on the graph seem to end with a small or zero trend (i. e. none seem close to convergence), suggesting that more time might reveal more basic block transitions and possibly a more credible difference of means.

For `listswf`, yet again, Bayesian estimation cannot provide us with enough confidence to say that the multiple winners strategy uncovers more basic block transitions than the union of fuzzers, as shown in [Figure A.8](#). Running the analysis to compare the results of the two cooperative strategies yields uncertainty in confirming that selecting multiple winners uncovers more basic block transitions than selecting

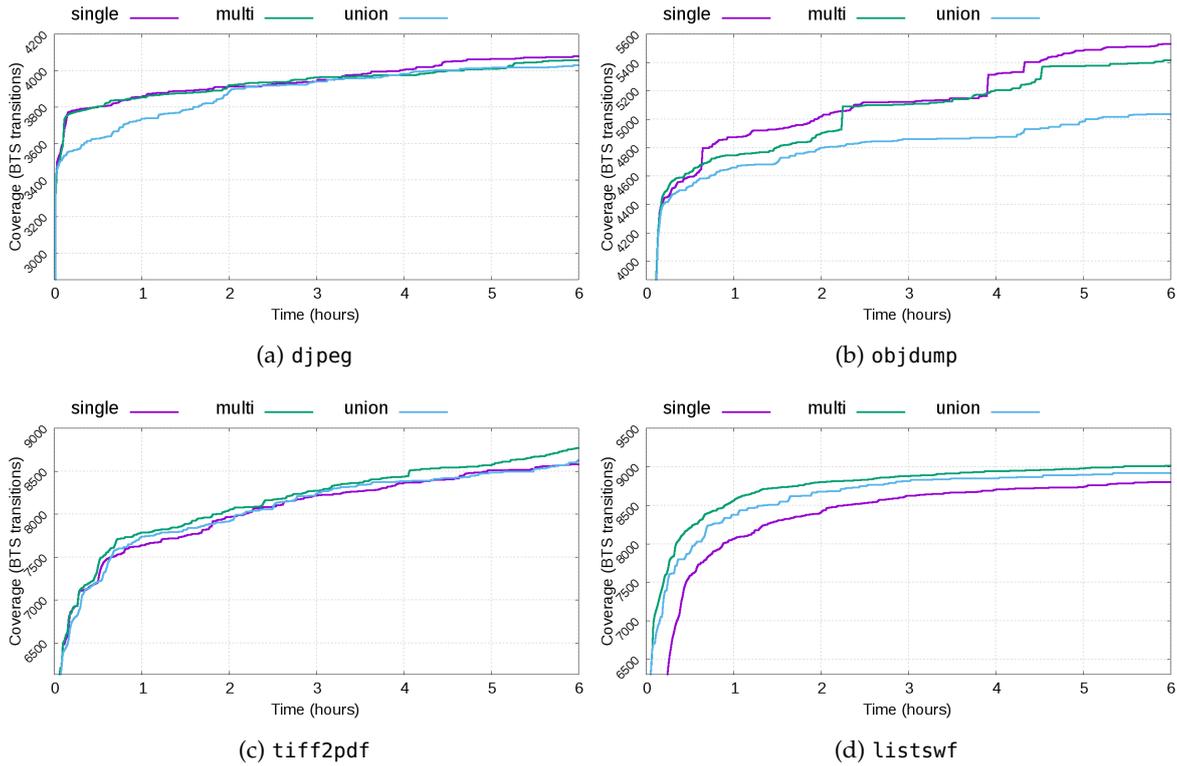


Figure 4.6: Mean coverage over time for two cooperative strategies and union of fuzzers.

a single one; this is shown in [Figure A.9](#) where we see that the HDI includes zero and 94% of the credible values are above zero.

4.3 CRASH ANALYSIS

This section presents our findings regarding crashes based on the experiments already described in [Section 4.2](#). In the following we compare the union of fuzzers and two cooperative strategies; in figures, “mono” (colour red) refers to the union, “multi” (green) and “single” (blue) refer to the multiple and single winner strategies respectively.

Unfortunately we were able to find crashes only in `listswf`; when discussing results, the remaining of the section does not refer to the SUT explicitly as there is only one to consider for this analysis.

Before presenting the results, we ought to define some terminology and the process that we used to obtain the final data. A *unique crash* is an input that causes a crash in the SUT through a unique

path. Each of the fuzzers we used in our experiments already track unique crashes, we developed some additional scripts to post-process the data and aggregate results of the five rounds we ran for each experiment. In particular, for each round of an experiment we collect all unique crashes from the fuzzer’s folder and run the SUT with the given file (we also disable ASLR and limit the address space appropriately). If the program terminates with a crash or timeouts (the time limit is taken from the respective fuzzer’s parameter), we run the GDB utility `exploitable` [46] and `backtrace` command and store the output into a file. The *stack hash* computed by `exploitable`, which is the hash of the last five calls on the stack, is used to identify a unique crash. Then, for each experiment, we aggregate the results of the rounds into a single file that contains the elapsed time at which the crash was discovered and the stack hash; moreover, we do not remove duplicates.

Table 4.5 shows the number of *distinct* unique crashes found across five rounds of each experiment, alongside the number of distinct crashes that were found by one experiment and not by the other.

	Unique crashes	Vs. single	Vs. multi	Vs. union
union	75	21	13	
multi	98	44		36
single	63		9	9

Table 4.5: Distinct unique crashes and amount discovered by one and not discovered by another.

Figure 4.7 presents the evolution over time of the unique crashes aggregated from the five rounds; in particular, Figure 4.7a shows the cumulative count of unique crashes, while Figure 4.7b shows their density.

With regards to the cumulative count, we see that not only the multiple winners strategy finds more distinct unique crashes (i. e. 98 against 75), but also the number of unique crashes itself is higher. Moreover, we see that the number of unique crashes for multiple winners is briefly taken over by that of the union of fuzzers, before

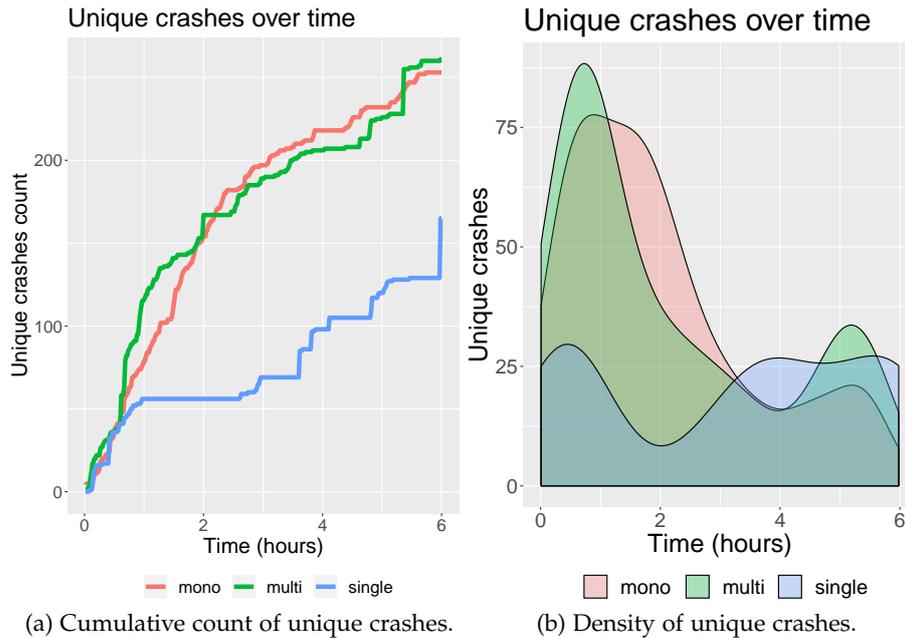


Figure 4.7: Unique crashes over time for listswf.

becoming the highest again toward the end of the runs; this is also highlighted by the strongly bi-modal nature of the density.

Figure 4.8 presents a subdivision of the results already presented in Figure 4.7b: Figure 4.8a shows the density of unique crashes only for those that fall in the intersection among all three experiments; Figure 4.8b shows the density for those that do not fall in the intersection.

From the figure, we see that for the intersection of crashes, the union of fuzzers produces the most amount and, with the help of the density of the remaining, we see that the multiple winners strategy's effort is spend producing more crashes that are not found by the other experiments.

4.3.1 Known Vulnerabilities

For each of the unique crashes, we manually investigated the nature of the crash and tried to link it to a known vulnerability with an assigned Common Vulnerabilities and Exposures (CVE) identifier; note that multiple unique crashes may be caused by the same bug. We

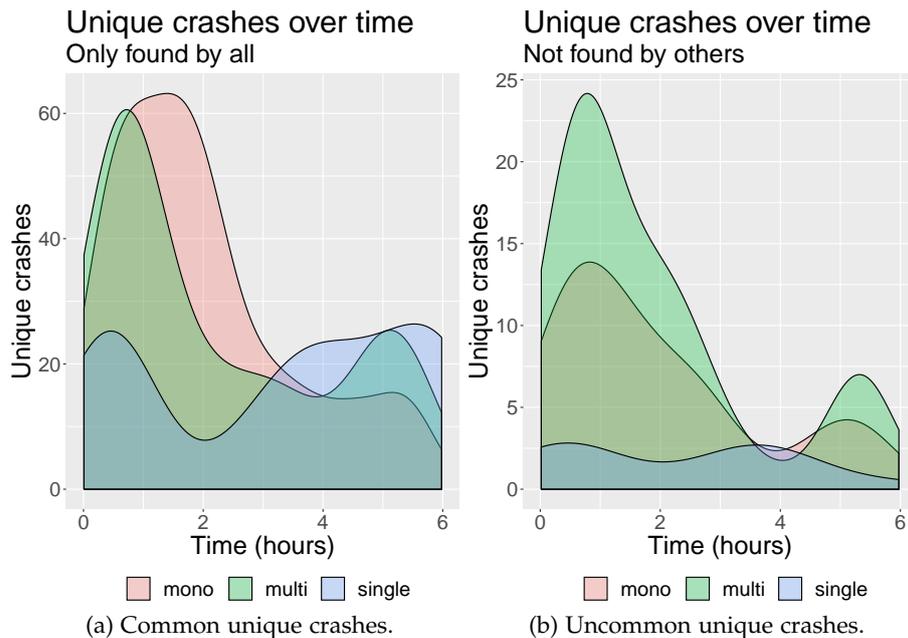


Figure 4.8: Density of unique crashes over time for listswf, divided in intersection of hashes and not in the intersection.

found exclusively memory access violations caused by unchecked heap memory allocations or reallocations, an example of which is given in Listing 4.1. We were able to link six (of the over 25 found) to a CVE; for those we report on the time of discovery in Figure 4.9.

```

char *readBytes(FILE *f,int size)
{
    int i;
    char *buf;

    buf = (char *)malloc(sizeof(char)*size);

    for(i=0;i<size;i++)
    {
        buf[i]=(char)readUInt8(f);
    }

    return buf;
}

```

Listing 4.1: Unchecked memory allocation in util/read.c:222 causing CVE-2017-7582.

We see that all CVEs found by the union of fuzzers have been found by the multiple winners strategy; moreover all but one are found on

average before the union does. The multiple winners strategy finds also two CVEs that are not discovered by the union of fuzzers.

4.4 OVERHEAD EVALUATION

In this final section we evaluate the overhead exercised by the operation of our implementation of the CFF over the regular operation of the underlying fuzzers. The fuzzers operation within the context of the CFF incur in some overhead due to the synchronization of external test cases injected from other fuzzers. AFLFast and FairFuzz inherit from AFL a mechanism that periodically checks for test cases to import; each new file is fed to the SUT and saved to the internal queue if deemed interesting. VUzzer, instead, synchronizes with inputs on the file system after each generation. As already mentioned in [Section 3.2.2](#), Honggfuzz had to be extended to allow for a synchronization mechanism similar to that of AFL; our extension periodically checks for new files in a given folder and directly imports them into its local queue.

The remaining overhead is given by the drivers and master operation. Recall from [Section 3.2.2](#) that a driver's responsibility is three-fold:

- checks for new interesting inputs via `inotify` and collects coverage information by running the SUT with BTS enabled before pushing a new message to the master;
- replies to metric requests by processing the attached BTS trace;
- receives new inputs from the master, copies the attached test case into the fuzzer's directory (to later be synced) and updates the coverage information with the attached BTS trace.

The master, recall from [Section 3.2.3](#), is responsible to collect new inputs from the drivers, request metric evaluations, select a winner (or multiple ones) and broadcast it via ZeroMQ. The average CPU usage after six hours for driver and master is respectively 2.63% and 0.18% without including the time spent by waited-for children

processes; with the inclusion of this, the CPU usage is 3.05% for each driver and is unchanged for the master (because the drivers run the SUT and wait for its completion). Note also that these measures are based on the amount of time the respective process has been scheduled and so it includes the time the process has spent idling.

The CPU usage of the master is clearly negligible; for the drivers instead, given that the CPU usage is only slightly higher with the inclusion of waited-for processes, we can deduce that the majority of resources are spent processing BTS traces and computing metrics instead of running the SUT.

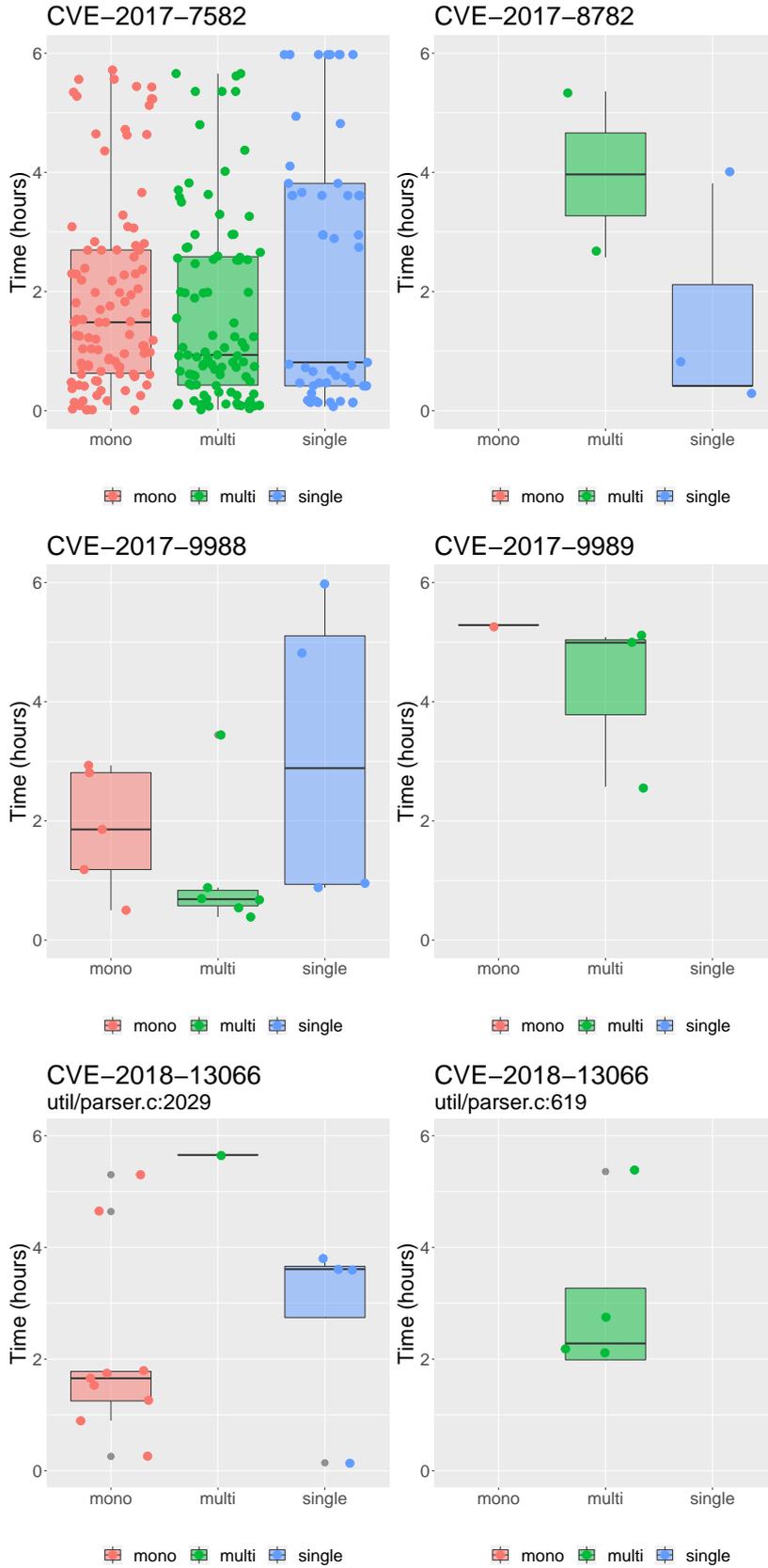


Figure 4.9: Bugs with assigned CVE identifier found in listswf.

DISCUSSION

This chapter contains a discussion of the evaluation results presented in [Chapter 4](#). With regards to our research question of whether for fuzzing the No Free Lunch Theorem holds, [Section 4.1](#) showed that some fuzzers perform decisively better than the other on some programs while perform poorly for other. This confirms our initial hypothesis that there are no free lunches for fuzzing.

In [Section 4.2](#) we tried to establish whether introducing cooperation into a group of fuzzers running in parallel proves beneficial. We tackled the question from two points of view: coverage and crashes found. When analyzing coverage results we were unable to declare a winner with credibly high certainty. Bayesian estimation could not provide strong evidence supporting the hypothesis that cooperation is beneficial; the results are nonetheless promising and one missing key ingredient to obtain a more decisive result, in our opinion, is more data. With more data, Bayesian estimation would return a picture of the difference of means that reflects more the true distribution, giving more space to the possibility to draw a confident conclusion. Moreover we are unable to find a decisive winner among the two cooperative strategies; although, we note that the multiple winners strategy performs better than the union of fuzzers even when the single winner strategy performs the best.

The analysis of crashes presented in [Section 4.3](#) reveals an image more in favour of cooperation. The cooperative strategy that uncovered the most basic block transitions uncovers also the most unique crashes. Furthermore the multiple winners cooperative strategy uncovers more distinct unique crashes than the union of fuzzers with a factor of 1.3. Also, it finds all CVEs that the union of fuzzers finds, plus two more that are not found by the union.

Besides doing more, longer runs, having data on a wider array of test programs would give a more generalized view on the subjects. In the process of developing the present work, we tried a number of other candidate programs which revealed to be problematic in a way or another:

`tcptrace-4.9.0` writes a lot to the temporary file system causing the OS performance to degrade;

`gif2png-2.5.11` writes its output in the same folder as its input, causing the fuzzer to use consider it a new input. VUzzer solves this with an optional parameter to filter input files by file extension; other fuzzers unfortunately don't offer this feature;

`xmllint` from `libxml2-2.9.4`. AFL-based fuzzers fail because unable to communicate with the fork server (recall we run in QEMU mode) and report a possible out-of-memory failure;

`tcpdump-4.9.0` encounters a known bug in `angr`¹ which is used by a pre-processing script in VUzzer. We experienced the same problem also with `nm` and `cxxfilt` from `binutils-2.28`;

`mutool` from `mupdf-1.9` fails in the same `angr` script because `claripy`, its constraint solver, is unable to minimize a constraint;

`mpg321-0.3.2` fails in VUzzer's pre-processing script because a node in the control-flow graph contains no instructions and the script expects at least one.

Finally, we turn our discussion toward the overhead evaluation presented in [Section 4.4](#) and add some performance considerations. Although no concrete data is available measuring the overhead of synchronization for each fuzzer, we note that the single source of overhead for AFL-based fuzzers is given by the execution of the SUT with the injected test case and in the decision of whether the test case is deemed interesting. In Honggfuzz the overhead is negligible as new

¹ <https://github.com/angr/angr/issues/288>

test cases are simply copied into the internal queue; a similar situation happens in VUzzer where the corpus is re-evaluated after each generation, picking injected test cases along newly generated inputs.

The remaining overhead given by drivers and master, reported respectively as 3.05% and 0.18% in average CPU usage, can be considered small compared to the normal operation of a fuzzer, which consistently uses one core at roughly 100%. Moreover note that the implementation of the CFF on which this evaluation is based, is not optimized with performance in mind and should be considered a prototype open to future improvements.

FUTURE WORK

In this chapter we discuss possible improvements to the CFF and its evaluation. To begin with, one possible line of research for future improvements may be to devise a more complex strategy that can exploit the BTS trace to the fullest. For example, one strategy could reason about the path exercised by a test case and compare it to the tree of already-discovered paths; a similarity metric can be returned to the master which will decide to which fuzzer to inject the test case. The similarity metric should be crafted so that higher values are associated to the exercised path which is closer to the discovered tree of the fuzzer. The intuition behind this is that the injection of such test cases may allow a fuzzer to better explore its vicinity. On the other hand one could devise a metric based on dissimilarity with the aim of injecting test cases that differ the most from the discovered tree to broaden the search and possibly escape local maxima.

Another possible extension to the present work is to broaden the spectrum of fuzzers used to evaluate the CFF. For example, the inclusion of a black box fuzzer wouldn't require any modification to the current implementation of the CFF to integrate it into the roster of fuzzers. A fuzzer of this kind would provide with much more throughput in terms of number of SUT executions given that no instrumentation is used; because of this though, a black box fuzzer is missing the notion of interesting input and its interaction with the CFF (only in the direction from the fuzzer to the driver) would need to be designed. To circumvent this, a black box mutational fuzzer may also be used only to inject new test cases (produced by other, more application-aware fuzzers) and not to extract them. The inclusion of a purely white box solution such as a symbolic execution engine could also yield interesting results. In this case, the CFF might integrate with it using the congestion control mechanism as has been

done for VUzzer (i. e. by injecting the test case with the highest score over a time window).

Finally, as already noted in [Chapter 5](#), the evaluation of the CFF would greatly benefit from having more than five runs for each experiment. This may allow for the Bayesian estimation to provide results on the difference of means that express less noise (i. e. more decisive results). On the same note, increasing the time length of the runs (e. g. from 6 hours to 24) would allow for the fuzzers to explore the SUT more and possibly reach and even more definite plateau in terms of coverage.

CONCLUSION

In this work we investigated the efficacy of running different fuzzers on different programs. Our hypothesis is that, due to the No Free Lunch theorem for optimization, no best fuzzer exists when their performance is averaged across all possible programs. To contrast this, we devise a Cooperative Fuzzing Framework (CFF) to allow communication and exchange of information between heterogeneous fuzzer instances running in parallel. We design a set of APIs that fuzzers must comply to in order for them to interface with our framework. These APIs, which are usually already implemented by most fuzzers, allow for *extraction* and *injection* of test cases from and into the fuzzer. A third, optional, interfacing method allows for a form of *congestion control* for slower fuzzers (e. g. those using heavier-weight analysis).

For every newly extracted test case, the framework evaluates it in terms of code coverage with regards to the already discovered execution tree for each of the running fuzzers (besides the one that found the test case); this evaluation consists in running the SUT with Intel BTS enabled in order to obtain an hardware-generated execution trace in the form of basic block transitions and then synthesizing a numeric value to represent this evaluation. A *central decisional unit* is responsible to collect these evaluations and reason about their returned values to come up with a set of fuzzers that are going to receive the given test case. The central decisional unit and the evaluation metric realize a specific *cooperative fuzzing strategy*.

We have implemented a prototype of the CFF that uses the number of newly discovered basic block transitions in a test case as evaluation metric and supports a configurable winning strategy (i. e. the component responsible to select the set of fuzzers that are going to receive the new test case). Four general winning strategies can be con-

figured: highest or lowest evaluation metric and higher or lower than a predetermined threshold.

In the evaluation, we presented the results of running four fuzzers without cooperation with the aim of proving our hypothesis of no best fuzzer. We ran five experiments of 24 hours long on each SUT — for four SUTs — and compared the average obtained coverage over time. The results of Bayesian estimation of the difference of means provided support to the confirmation of our hypothesis as, for all considered SUTs, the best performing fuzzer is always different.

Then, we evaluated the effect of introducing cooperation by showing results of running our implementation of the CFF for 6 hours on the same SUTs. We configured the framework to evaluate two strategies: one selects a single winner with the highest metric, the other selects all fuzzers that have returned a metric higher than zero. In the experiments we used AFLFast, FairFuzz, Honggfuzz and VUzzer as fuzzer components to the CFF. We compared the results from the two cooperative strategies with the union of the results from running the same four fuzzers without cooperation. Unfortunately, although the final mean coverage is higher for all programs for one of the cooperative strategies, we cannot claim a definitive winner as Bayesian estimation of the difference of means does not support it with strong confidence, revealing instead a need for more data.

Moreover, we compared the capability of finding unique crashes and known vulnerabilities of the CFF and the union of fuzzers. In particular, the cooperative strategy that uncovers the most basic block transitions also finds more unique crashes than the union of fuzzers with a factor of 1.3. Furthermore, the CFF finds all the CVEs that the union finds, plus two more.

Lastly, we provided an overhead evaluation in which we note that the resources consumed by the components of the CFF are not excessive compared to the normal operation of its fuzzers. This is especially true, given that the present implementation has not been extensively developed with performance in mind and should be considered a prototype.

BAYESIAN ESTIMATION OF COOPERATIVE STRATEGIES

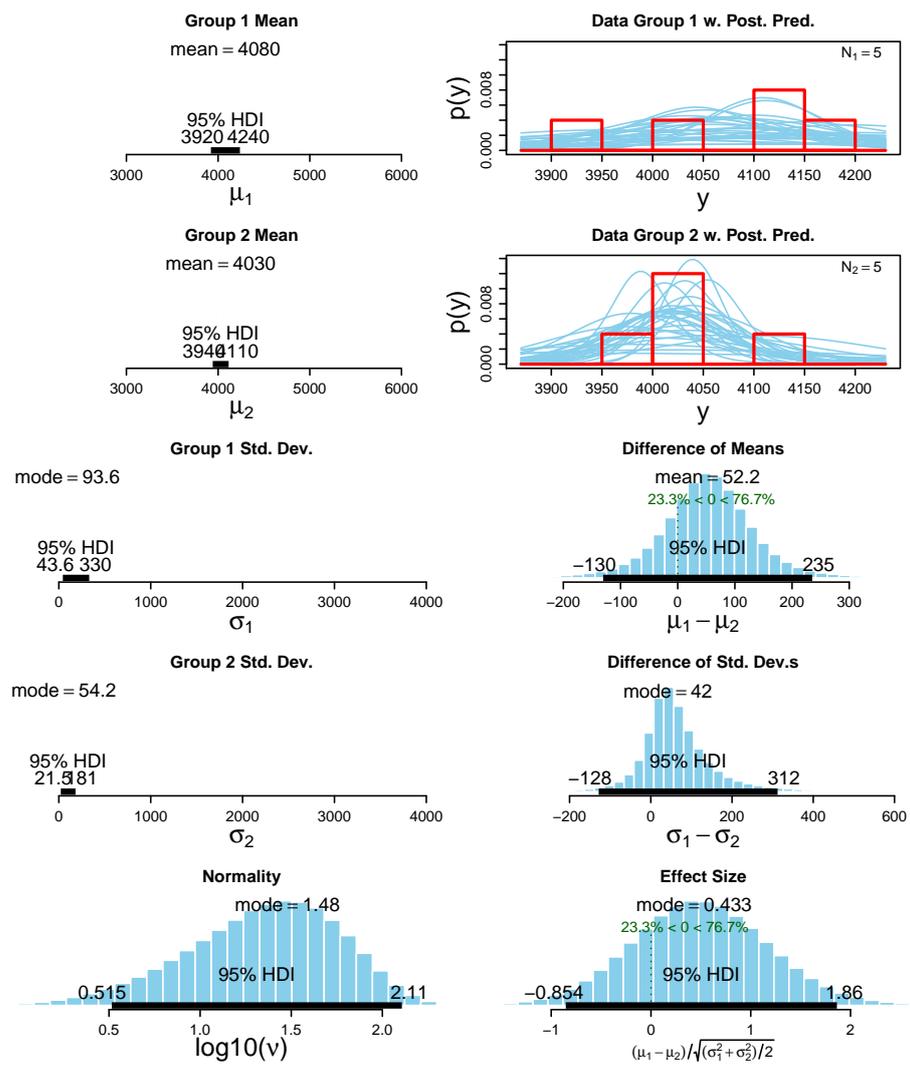


Figure A.1: Bayesian estimation for single winner strategy vs. union of fuzzers for jpeg.

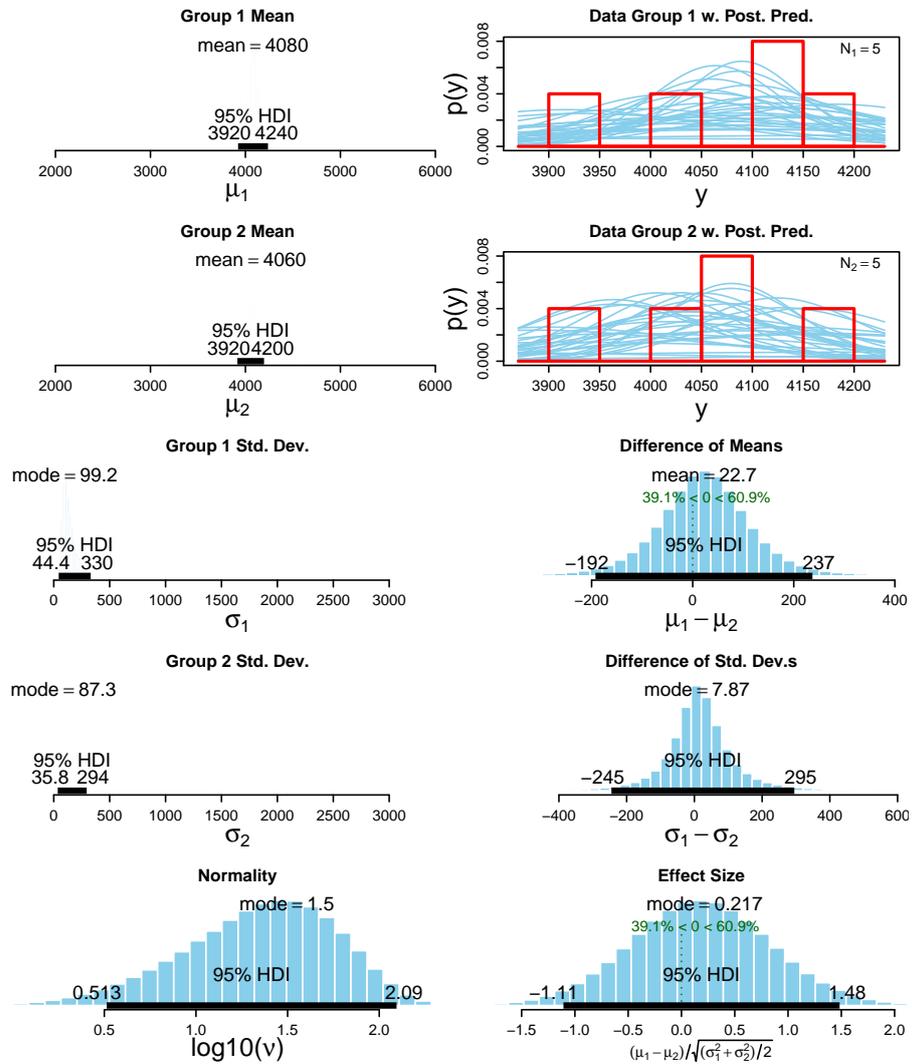


Figure A.2: Bayesian estimation for single winner strategy vs. multiple winners strategy for jpeg.

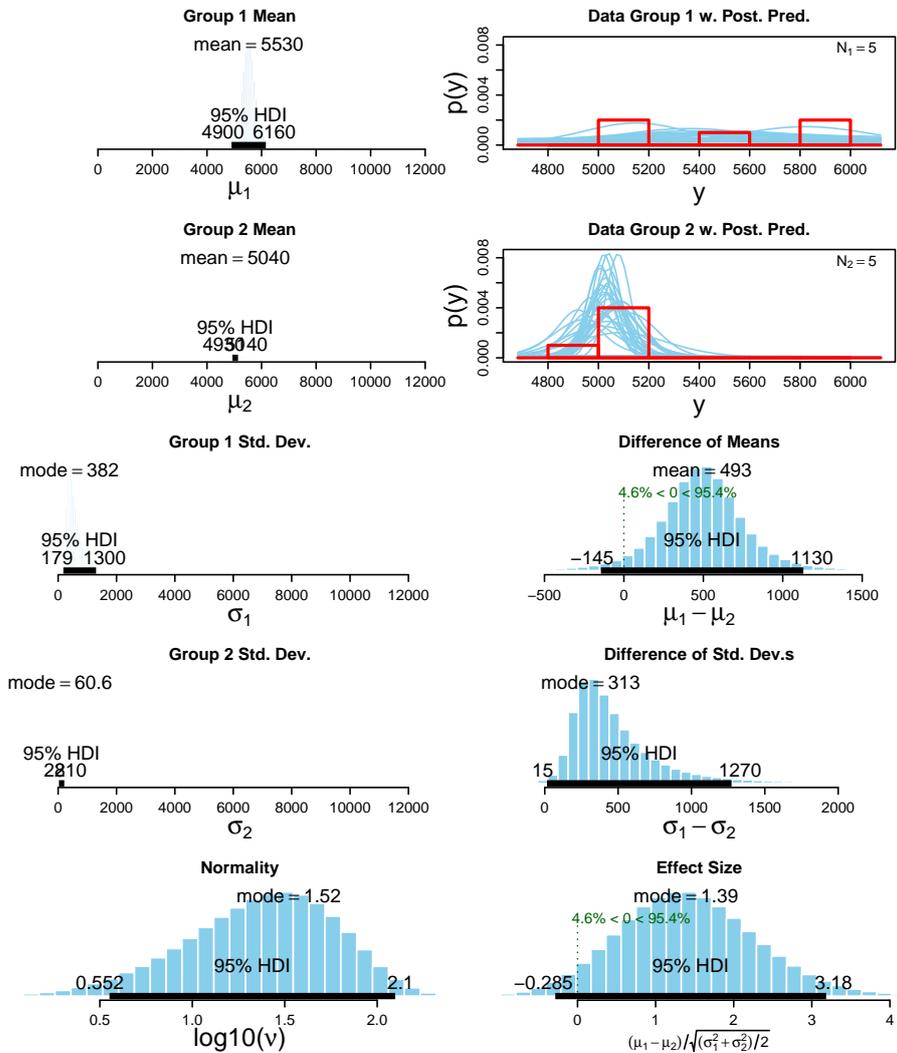


Figure A.3: Bayesian estimation for single winner strategy vs. union of fuzzers for objdump.

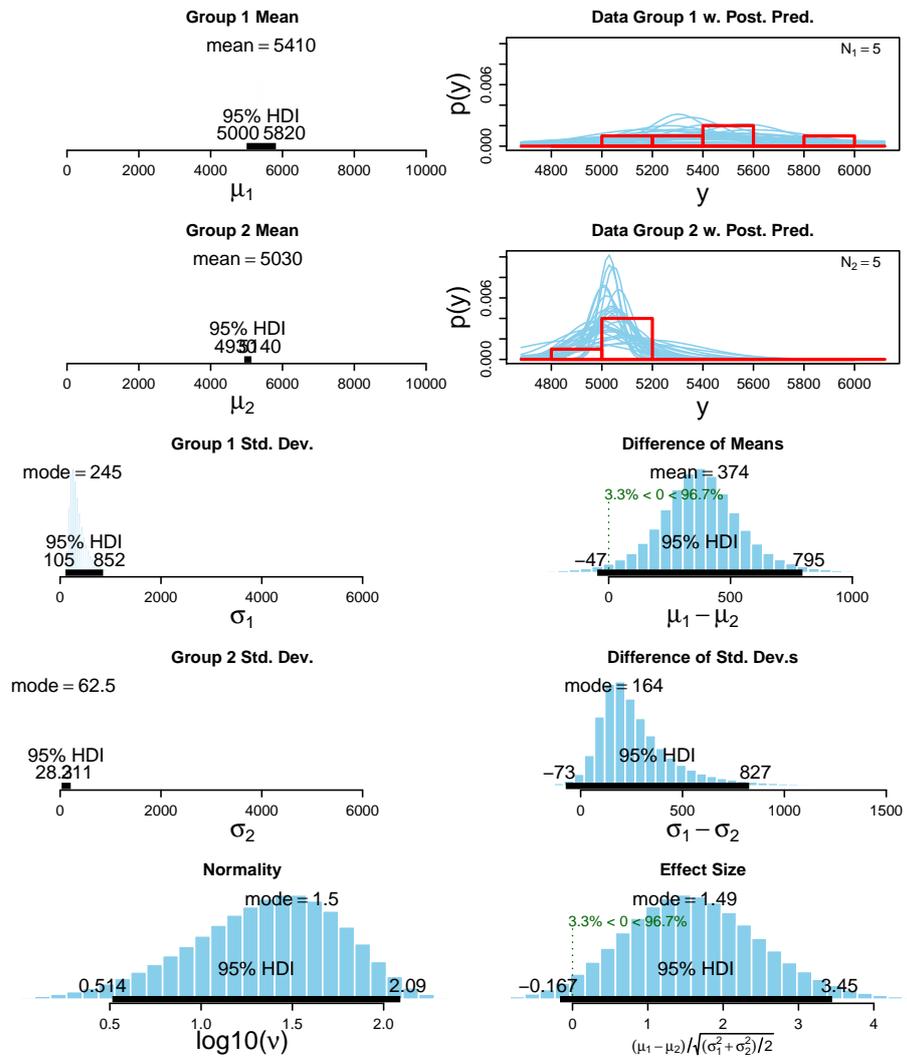


Figure A.4: Bayesian estimation for multiple winners strategy vs. union of fuzzers for objdump.

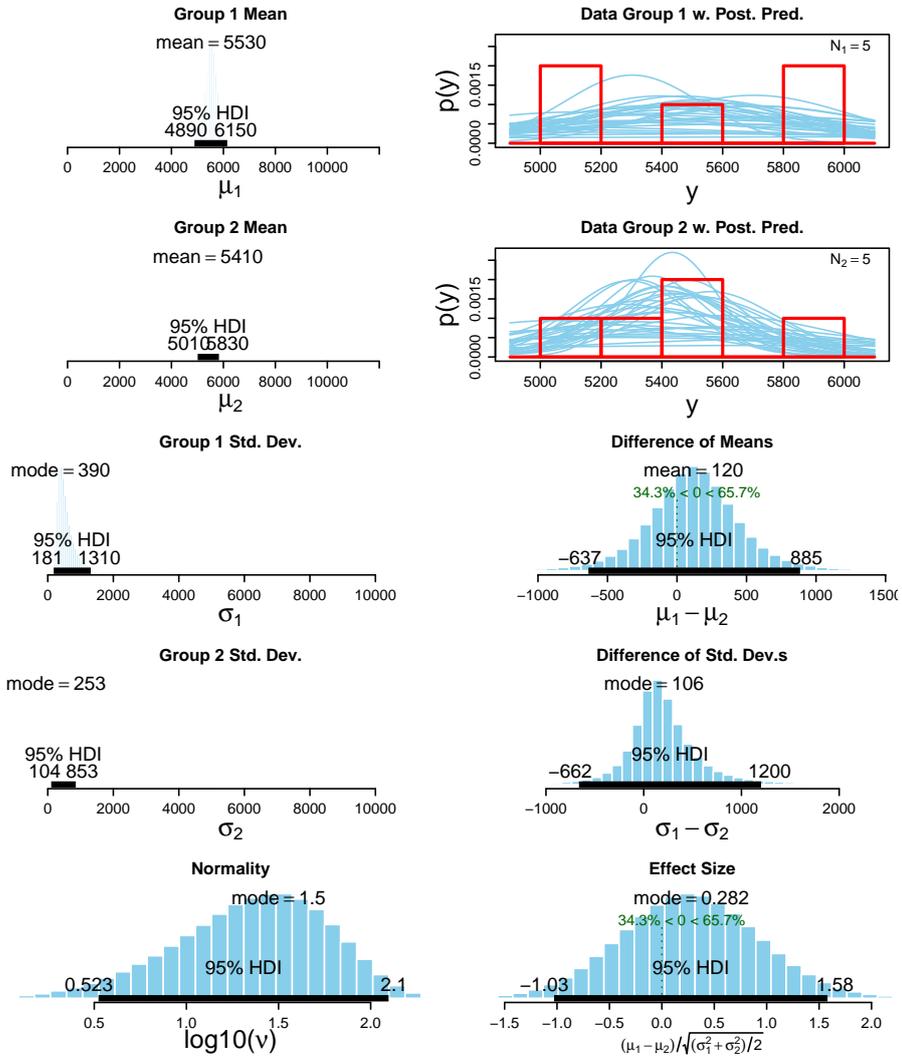


Figure A.5: Bayesian estimation for single winner strategy vs. multiple winners strategy for objdump.

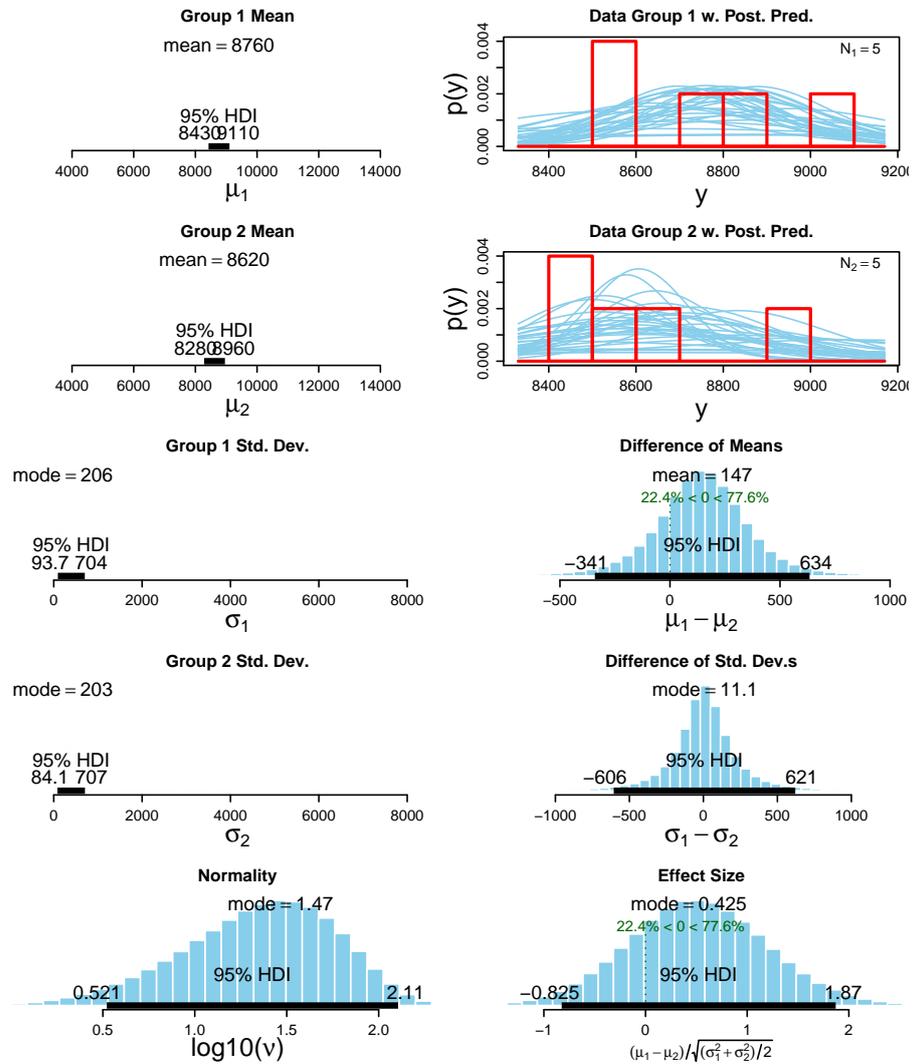


Figure A.6: Bayesian estimation for multiple winners strategy vs. union of fuzzers for tiff2pdf.

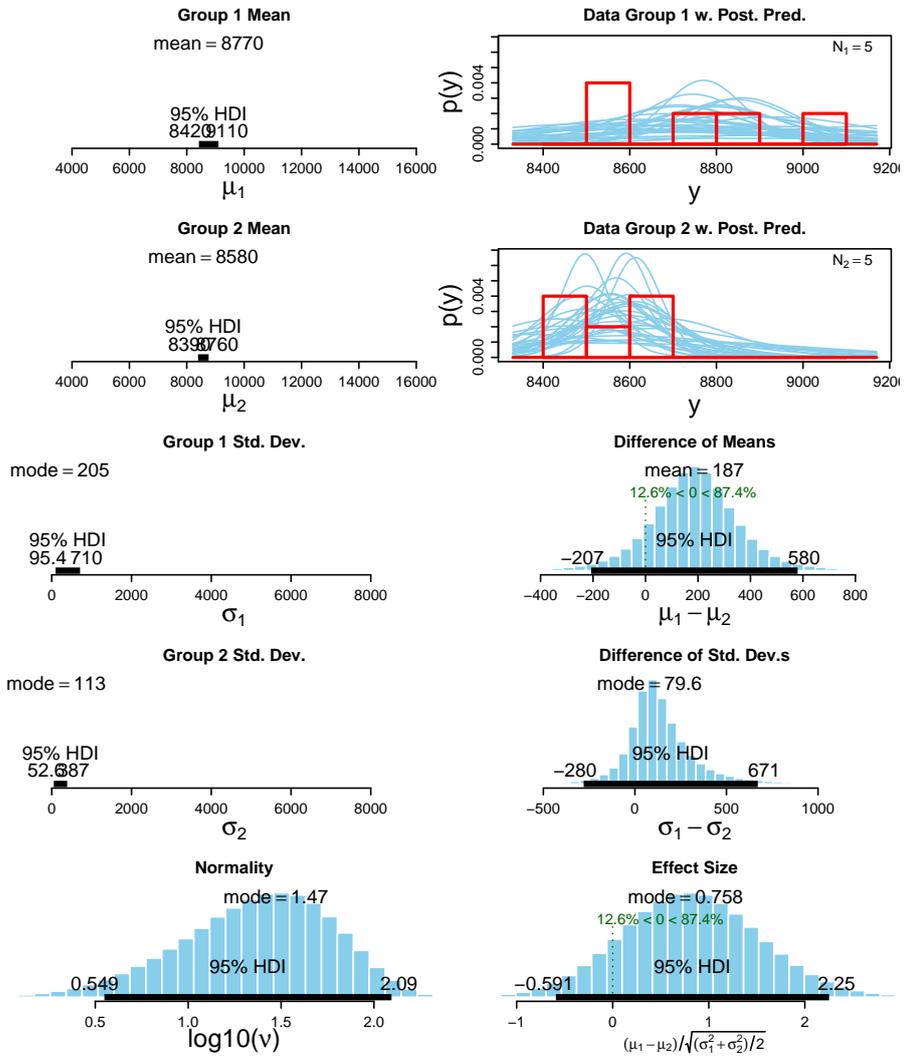


Figure A.7: Bayesian estimation for multiple winners strategy vs. single winner strategy for tiff2pdf.

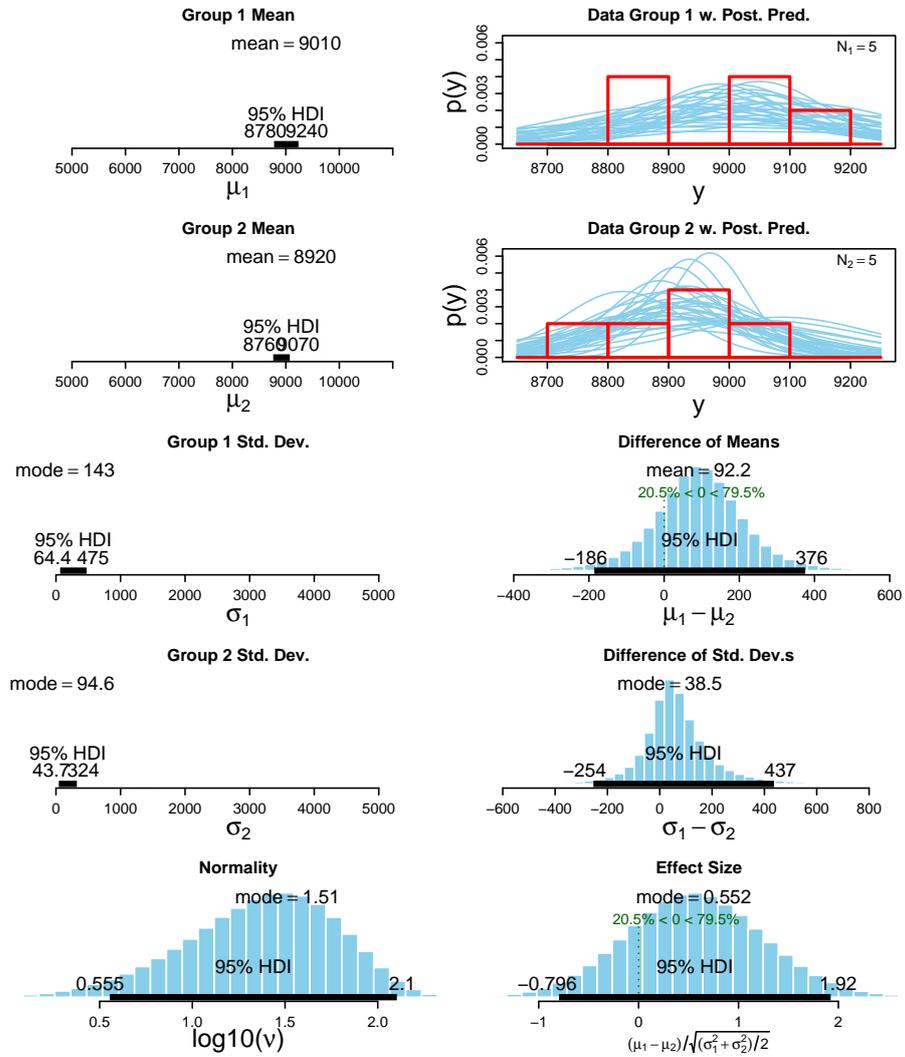


Figure A.8: Bayesian estimation for multiple winners strategy vs. union of fuzzers for listswf.

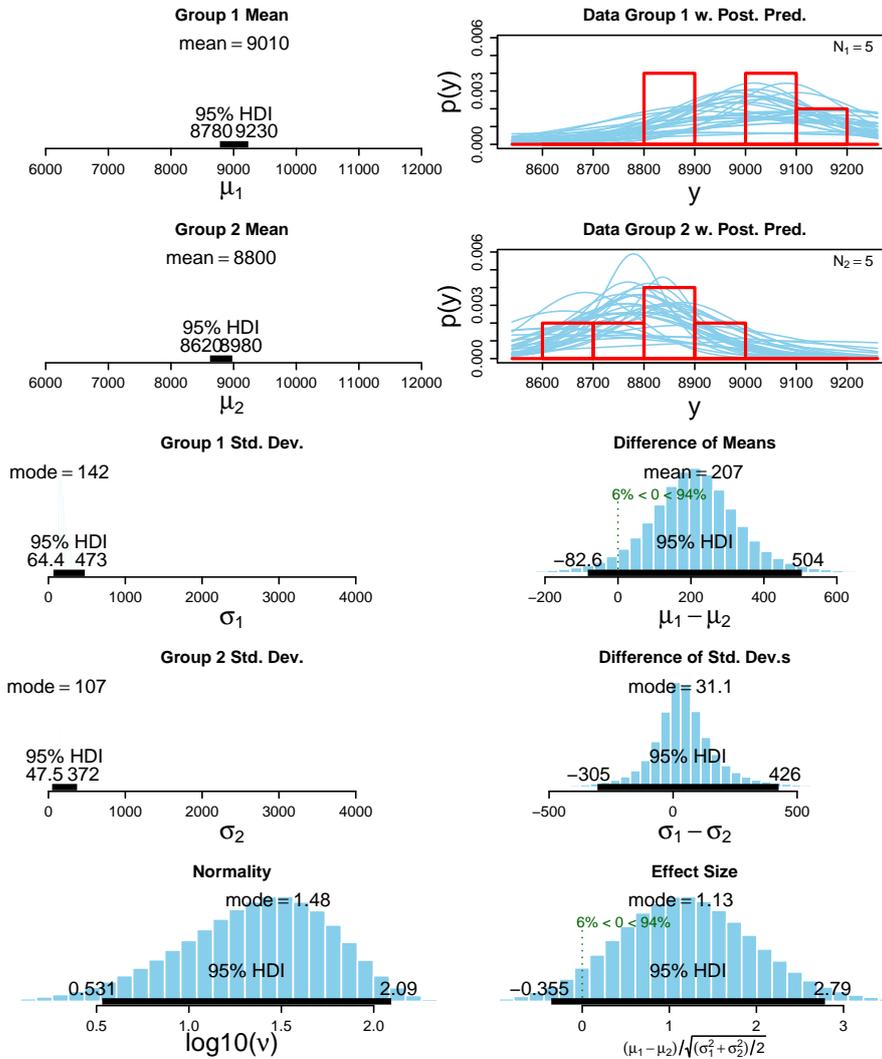


Figure A.9: Bayesian estimation for multiple winners strategy vs. single winner strategy for listswf.

BIBLIOGRAPHY

- [1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. *Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software*. Dec. 2016. URL: <https://security.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html>.
- [2] Frances E. Allen. "Control Flow Analysis." In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19. DOI: [10.1145/800028.808479](https://doi.org/10.1145/800028.808479). URL: <http://doi.acm.org/10.1145/800028.808479>.
- [3] Frances E Allen and John Cocke. *Graph-theoretic constructs for program control flow analysis*. IBM Thomas J. Watson Research Center, 1972.
- [4] *American Fuzzy Lop + Dyninst == AFL Fuzzing blackbox binaries*. URL: <https://github.com/vrtadmin/moflow/tree/master/afl-dyninst>.
- [5] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.
- [6] David Aspinall and Martin Hofmann. "Dependent Types." In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. The MIT Press, 2004. Chap. 2, pp. 45–86. ISBN: 0262162288.
- [7] Lennart Augustsson. "Cayenne—A Language with Dependent Types." In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: ACM, 1998, pp. 239–250. ISBN: 1-58113-024-4. DOI: [10.1145/289423.289451](https://doi.org/10.1145/289423.289451). URL: <http://doi.acm.org/10.1145/289423.289451>.

- [8] Alberto Bacchelli and Christian Bird. "Expectations, outcomes, and challenges of modern code review." In: *Proceedings of the 2013 international conference on software engineering*. IEEE Press. 2013, pp. 712–721.
- [9] James Bach and Patrick J Schroeder. "Pairwise testing: A best practice that isn't." In: *Proceedings of 22nd Pacific Northwest Software Quality Conference*. Citeseer. 2004, pp. 180–196.
- [10] Greg Banks, Marco Cova, Viktoria Felmetzger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. "SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr." In: *International Conference on Information Security*. Springer. 2006, pp. 343–358.
- [11] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The oracle problem in software testing: A survey." In: *IEEE transactions on software engineering* 41.5 (2015), pp. 507–525.
- [12] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. "Synthesizing program input grammars." In: *ACM SIGPLAN Notices*. Vol. 52. 6. ACM. 2017, pp. 95–110.
- [13] Mike Batongbacal and Syed Medhi. *Microsoft Security Risk Detection*. May 2017. URL: <https://devblogs.microsoft.com/premier-developer/microsoft-security-risk-detection/>.
- [14] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004. ISBN: 0321278658.
- [15] Boris Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. New York, NY, USA: John Wiley & Sons, Inc., 1995. ISBN: 0-471-12094-4.
- [16] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.
- [17] Donald A Berry and Bert Fristedt. "Bandit problems: sequential allocation of experiments (Monographs on statistics and

- applied probability)." In: *London: Chapman and Hall* 5 (1985), pp. 71–87.
- [18] William Blum. *Neural fuzzing: applying DNN to software security testing*. Nov. 2017. URL: <https://www.microsoft.com/en-us/research/blog/neural-fuzzing/>.
- [19] Barry W. Boehm. "Verifying and validating software requirements and design specifications." In: *IEEE software* 1.1 (1984), p. 75.
- [20] Marcel Böhme and Soumya Paul. "A probabilistic analysis of the efficiency of automated software testing." In: *IEEE Transactions on Software Engineering* 42.4 (2016), pp. 345–360.
- [21] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based greybox fuzzing as markov chain." In: *IEEE Transactions on Software Engineering* (2017).
- [22] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed greybox fuzzing." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 2329–2344.
- [23] Konstantin Böttinger. "Chemotactic Test Case Recombination for Large-Scale Fuzzing." In: *Journal of Cyber Security* 5.4 (), pp. 269–286.
- [24] Konstantin Böttinger. "Fuzzing binaries with Lévy flight swarms." In: *EURASIP Journal on Information Security* 2016.1 (2016), p. 28.
- [25] Konstantin Böttinger. "Hunting bugs with Lévy flight foraging." In: *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2016, pp. 111–117.
- [26] Konstantin Böttinger. "Guiding a colony of black-box fuzzers with chemotaxis." In: *2017 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2017, pp. 11–16.
- [27] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT—a formal system for testing and debugging programs by symbolic execution." In: 1975.

- [28] Edwin C. Brady. "Idris — systems programming meets full dependent types." In: *In Proc. 5th ACM workshop on Programming languages meets program verification, PLPV '11*. ACM, 2011, pp. 43–54.
- [29] Melvin A Breuer. "A random and an algorithmic technique for fault detection test generation for sequential circuits." In: *IEEE Transactions on Computers* 100.11 (1971), pp. 1364–1370.
- [30] Jacob Burnim and Koushik Sen. "Heuristics for Scalable Dynamic Test Generation." In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (2008), pp. 443–446.
- [31] John N Buxton and Brian Randell. *Software engineering techniques: report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*. NATO Science Committee, 1970.
- [32] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.
- [33] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. "EXE: automatically generating inputs of death." In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008), p. 10.
- [34] L Cardelli. "Typeful Programming. Formal Descriptions of Programming Concepts." In: *Springer-Verlag, Berlin/New York* 45 (1991), p. 1989.
- [35] Luca Cardelli. "Type Systems." In: *Computer Science Handbook, Second Edition*. Ed. by Allen B. Tucker. Chapman & Hall/CRC, 2004. ISBN: 158488360X.
- [36] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing mayhem on binary code." In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.

- [37] Oliver Chang, Abhishek Arya, Kostya Serebryany, and Josh Armour. *OSS-Fuzz: Five months later, and rewarding projects*. May 2017. URL: <https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html>.
- [38] Tsong Yueh Chen, Hing Leung, and I. K. Mak. "Adaptive Random Testing." In: *ASIAN*. 2004.
- [39] Lori A. Clarke. "A System to Generate Test Data and Symbolically Execute Programs." In: *IEEE Transactions on Software Engineering* SE-2 (1976), pp. 215–222.
- [40] Philip B Crosby. "The art of making quality certain." In: *New York: New American Library* 17 (1979).
- [41] Jacek Czerwonka. "Pairwise testing in real world." In: *24th Pacific Northwest Software Quality Conference*. Vol. 200. Citeseer. 2006.
- [42] Giuseppe A Di Lucca and Anna Rita Fasolino. "Testing Web-based applications: The state of the art and future trends." In: *Information and Software Technology* 48.12 (2006), pp. 1172–1186.
- [43] Edsger Wybe Dijkstra et al. *Notes on structured programming*. 1970.
- [44] Chris Evans, Matt Moore, and Ormandy Travis. *Fuzzing at scale*. Aug. 2011. URL: <https://security.googleblog.com/2011/08/fuzzing-at-scale.html>.
- [45] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. "Development and deployment at Facebook." In: *IEEE Internet Computing* 17.4 (2013), pp. 8–17.
- [46] Jonathan Foote. *Cert triage tools*. 2013. URL: <https://github.com/jfoote/exploitable>.
- [47] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [48] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

- [49] Patrice Godefroid. "Random testing for security: blackbox vs. whitebox fuzzing." In: *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM. 2007.
- [50] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-based whitebox fuzzing." In: *ACM Sigplan Notices*. Vol. 43. 6. ACM. 2008, pp. 206–215.
- [51] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *PLDI*. 2005.
- [52] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated Whitebox Fuzz Testing." In: *NDSS*. Vol. 8. 2008, pp. 151–166.
- [53] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing." In: *Communications of the ACM* 55.3 (2012), pp. 40–44.
- [54] 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.
- [55] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390069X, 9780133900699.
- [56] Mats Grindal, Jeff Offutt, and Sten F Andler. "Combination testing strategies: a survey." In: *Software Testing, Verification and Reliability* 15.3 (2005), pp. 167–199.
- [57] Anthony Hall. "Seven Myths of Formal Methods." In: *IEEE Softw.* 7.5 (Sept. 1990), pp. 11–19. ISSN: 0740-7459. DOI: [10.1109/52.57887](https://doi.org/10.1109/52.57887). URL: <https://doi.org/10.1109/52.57887>.
- [58] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321154916.

- [59] Aki Helin. *Radamsa fuzzer*. 2015. URL: <https://github.com/aoh/radamsa>.
- [60] S Hocevar. *zzuf—multi-purpose fuzzer*. 2011. URL: <http://caca.zoy.org/wiki/zzuf>.
- [61] Allen D Householder and Jonathan M Foote. *Probability-based parameter selection for black-box fuzz testing*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2012.
- [62] William E Howden. “Theoretical and empirical studies of program testing.” In: *IEEE Transactions on Software Engineering* 4 (1978), pp. 293–298.
- [63] “IEEE Standard for System, Software, and Hardware Verification and Validation.” In: *IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/Incorporates IEEE Std 1012-2016/Cor1-2017)* (2017), pp. 1–260. DOI: [10.1109/IEEESTD.2017.8055462](https://doi.org/10.1109/IEEESTD.2017.8055462).
- [64] Ann Johnson. *Application fuzzing in the era of Machine Learning and AI*. Jan. 2018. URL: <https://www.microsoft.com/security/blog/2018/01/03/application-fuzzing-in-the-era-of-machine-learning-and-ai/>.
- [65] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
- [66] Cem Kaner. “A tutorial in exploratory testing.” In: *Tutorial presented at QUEST2008* (2008).
- [67] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software Second Edition*. Dreamtech Press, 2000.
- [68] Niall Kennedy. *Google Mondrian: web-based code review and storage*. 2006. URL: <https://www.niallkennedy.com/blog/2006/11/google-mondrian.html>.
- [69] Brian W. Kernighan. *The C Programming Language*. Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.
- [70] James C. King. “Symbolic Execution and Program Testing.” In: *Commun. ACM* 19 (1976), pp. 385–394.

- [71] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel." In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: [10.1145/1629575.1629596](https://doi.acm.org/10.1145/1629575.1629596). URL: <http://doi.acm.org/10.1145/1629575.1629596>.
- [72] Bogdan Korel. "Automated Software Test Data Generation." In: *IEEE Trans. Software Eng.* 16 (1990), pp. 870–879.
- [73] John K Kruschke. "Bayesian estimation supersedes the t test." In: *Journal of Experimental Psychology: General* 142.2 (2013), p. 573.
- [74] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. "Software fault interactions and implications for software testing." In: *IEEE transactions on software engineering* 30.6 (2004), pp. 418–421.
- [75] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient state merging in symbolic execution." In: *In Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*. 2012.
- [76] Yu Lei and Kuo-Chung Tai. "In-parameter-order: A test generation strategy for pairwise testing." In: *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*. IEEE. 1998, pp. 254–261.
- [77] Caroline Lemieux and Koushik Sen. "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage." In: *arXiv preprint arXiv:1709.07101* (2017).
- [78] Steve Lipner. "The trustworthy computing security development lifecycle." In: *Computer Security Applications Conference, 2004. 20th Annual*. IEEE. 2004, pp. 2–13.
- [79] 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 anal-

- ysis tools with dynamic instrumentation.” In: *Acm sigplan notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.
- [80] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. “Directed Symbolic Execution.” In: *Proceedings of the 18th International Conference on Static Analysis. SAS’11*. Venice, Italy: Springer-Verlag, 2011, pp. 95–111. ISBN: 978-3-642-23701-0. URL: <http://dl.acm.org/citation.cfm?id=2041552.2041563>.
- [81] Rupak Majumdar and Koushik Sen. “Hybrid concolic testing.” In: *29th International Conference on Software Engineering (ICSE’07)*. IEEE. 2007, pp. 416–426.
- [82] Robert Mandl. “Orthogonal Latin squares: an application of experiment design to compiler testing.” In: *Communications of the ACM* 28.10 (1985), pp. 1054–1058.
- [83] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.
- [84] Nicholas D. Matsakis and Felix S. Klock II. “The Rust Language.” In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: [10.1145/2692956.2663188](https://doi.org/10.1145/2692956.2663188). URL: <http://doi.acm.org/10.1145/2692956.2663188>.
- [85] Barton P Miller, Louis Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities.” In: *Communications of the ACM* 33.12 (1990), pp. 32–44.
- [86] Max Moroz and Kostya Serebryany. *Guided in-process fuzzing of Chrome components*. Aug. 2016. URL: <https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html>.
- [87] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.
- [88] Hung Q Nguyen. *Testing applications on the Web: Test planning for Internet-based systems*. John Wiley & Sons, 2001.

- [89] Brian S Pak. "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution." In: *School of Computer Science Carnegie Mellon University* (2012).
- [90] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "Model-based whitebox fuzzing for program binaries." In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 543–553.
- [91] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "Vuzzer: Application-aware evolutionary fuzzing." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017.
- [92] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. "Optimizing seed selection for fuzzing." In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 861–875.
- [93] Debra J Richardson and Lori A Clarke. "A partition analysis method to increase program reliability." In: *Proceedings of the 5th international conference on Software engineering*. IEEE Press. 1981, pp. 244–253.
- [94] "SO/IEC/IEEE Draft Standard for Software and Systems Engineering–Software Testing–Part 1: Concepts and Definitions." In: *ISO/IEC/IEEE P29119-1/DIS, September 2012* (2012), pp. 1–64.
- [95] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. "Have things changed now? An empirical study on input validation vulnerabilities in web applications." In: *Computers & Security* 31.3 (2012), pp. 344–356.
- [96] Koushik Sen and Gul A. Agha. "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools." In: *CAV*. 2006.
- [97] Koushik Sen, Darko Marinov, and Gul A. Agha. "CUTE: a concolic unit testing engine for C." In: 2005.

- [98] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. "Sok:(state of) the art of war: Offensive techniques in binary analysis." In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.
- [99] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWE-BOK(R)): Version 3.0*. 3rd. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014. ISBN: 0769551661, 9780769551661.
- [100] Matt Staats and Corina Păsăreanu. "Parallel Symbolic Execution for Structural Test Generation." In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. Trento, Italy: ACM, 2010, pp. 183–194. ISBN: 978-1-60558-823-0. DOI: [10.1145/1831708.1831732](https://doi.org/10.1145/1831708.1831732). URL: <http://doi.acm.org/10.1145/1831708.1831732>.
- [101] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS*. Vol. 16. 2016, pp. 1–16.
- [102] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013. ISBN: 0321563840, 9780321563842.
- [103] Nikhil Swamy et al. "Dependent Types and Multi-Monadic Effects in F*." In: *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. URL: <https://www.fstar-lang.org/papers/mumon/>.
- [104] Robert Swiecki. *Honggfuzz*. URL: <https://github.com/google/honggfuzz>.
- [105] Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

- [106] Keizo Tatsumi. "Test case design support system." In: *Proc. International Conference on Quality Control (ICQC'87)*. 1987, pp. 615–620.
- [107] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection." In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 497–512.
- [108] Gerald M. Weinberg. *Perfect Software: And Other Illusions About Testing*. New York, NY, USA: Dorset House Publishing Co., Inc., 2008. ISBN: 0932633692, 9780932633699.
- [109] David H Wolpert, William G Macready, et al. "No free lunch theorems for optimization." In: *IEEE transactions on evolutionary computation* 1.1 (1997), pp. 67–82.
- [110] 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 & communications security*. ACM. 2013, pp. 511–522.
- [111] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. "Formal Methods: Practice and Experience." In: *ACM Comput. Surv.* 41.4 (Oct. 2009), 19:1–19:36. ISSN: 0360-0300. DOI: [10.1145/1592434.1592436](https://doi.org/10.1145/1592434.1592436). URL: <http://doi.acm.org/10.1145/1592434.1592436>.
- [112] Hongwei Xi. "Dependent ML An approach to practical programming with dependent types." In: *Journal of Functional Programming* 17.2 (2007), pp. 215–286.
- [113] Hongwei Xi and Frank Pfenning. "Dependent types in practical programming." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1999, pp. 214–227.
- [114] Michal Zalewski. *American Fuzzy Lop*. URL: <http://lcamtuf.coredump.cx/afl/>.

- [115] Michal Zalewski. *Binary fuzzing strategies: what works, what doesn't*. Aug. 2014. URL: <https://lcamtuf.blogspot.nl/2014/08/binary-fuzzing-strategies-what-works.html>.
- [116] Michal Zalewski. *Pulling JPEGs out of thin air*. Nov. 2014. URL: <https://lcamtuf.blogspot.nl/2014/11/pulling-jpegs-out-of-thin-air.html>.
- [117] Michal Zalewsky. *Technical "whitepaper" for afl-fuzz*. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt.