SAPIENZA
UNIVERSITÀ DI ROMA

Program State Abstraction for Feedback-Driven Fuzz Testing using Likely Invariants

Faculty of Information Engineering, Computer Science and Statistics

Degree course in Engineering in Computer Science

Candidate

Andrea Fioraldi
ID number 1692419

Thesis Advisor	Co-Advisor
Dr. Daniele Cono D'Elia	Prof. Davide Balzarotti

Academic Year 2019/2020

Program State Abstraction for Feedback-Driven Fuzz Testing using Likely Invariants
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: andreafioraldi@gmail.com

To my grandmother L.,
who when she was a brilliant child,
despite the darkest hours of our country had just passed,
could not continue studying because the hearts of many were still black.

Abstract

Fuzz testing proved its great effectiveness in finding software bugs in the latest years, however, there are still open challenges. Coverage-guided fuzzers suffer from the fact that covering a program point does not ensure the trigger of a fault. Other more sensitive techniques that in theory should cope with this problem, such as the coverage of the memory values, easily lead to path explosion. In this thesis, we propose a new feedback for Feedback-driven Fuzz testing that combines code coverage with the "shape" of the data. We learn likely invariants for each basic block in order to divide into regions the space described by the variables used in the block. The goal is to distinguish in the feedback when a block is executed with values that fall in different regions of the space. This better approximates the program state coverage and, on some targets, improves the ability of the fuzzer in finding faults. We developed a prototype using LLVM and AFL++ called InvsCov.

Acknowledgments

I want to thank my advisors Davide and Daniele for the work done, in temporal order, to support me in the journey of the master thesis, despite a global pandemic and a busy student (myself) with too many other projects to carry on. I want to thank my family, my old and new friends, and the hackers' community that always I'm proud to be part of for the support and encouragement to pursue my work. This thesis was made mostly during my internship in the S3 lab of EURECOM, that I want to thank for the funding, the burned CPU cores, and the nice welcome.

Contents

Chapter 1

Introduction

In the last two decades *Fuzz Testing* (or *Fuzzing*) gained popularity thanks to its ability in finding software bugs more effectively than other Software Testing techniques.

It is employed every day since 2016 in Google's OSS-FUZZ [5] to continuously discover vulnerabilities in open source software and thousands of them were discovered so far in this four years of activity of the program.

This popularity comes also with the attention of academia and the industry on improving fuzzing techniques.

In recent times, on top of the twist of fuzz testing called *Coverage-guided Fuzz Testing (CGF)*, several new techniques were developed trying to overcome the limitations of CGF like hard to bypass path constraints [98] [81] [90] [84] [11] or the high number of invalid testcases generated by mutation [79] [74] [7] [16] [35].

These improvements, with some exceptions like [73], try to improve the ability of a fuzzer to reach more code coverage. Code coverage is used as a proxy to *Program State Coverage* to avoid path explosion because the number of program states can be potentially infinite. *Symbolic Execution* [13] for instance does not employ such approximation, and in fact, path explosion is one of the most critical problems that make pure symbolic-based approaches impractical in real-world targets.

There is empirical evidence that fuzzers that uncover more code coverage discover also more bugs in programs. This can be motivated by the observation that exploring a portion of code is a necessary condition to find a fault in that code portion.

However, this is not a sufficient condition.

Fuzzers can saturate in coverage and never reach the combination of program states that leads to a bug. To cope with this problem, a fuzzer should observe the progress also in the program state data, not only in the control flow.

Recent works like [75] [10] [95] try to go beyond simple code coverage as feedback; we refer to these techniques — CGF included — as *Feedback-driven Fuzz Testing.*

Some fuzzers approximate the program state using more sensitive feedbacks, like code coverage with call stack information or even code coverage and values loaded and stored from memory. This second approach, as shown by [95], better approximates the program state coverage by taking into account not only control flow but also the values in the program state data, but it is less efficient in finding bugs because of path explosion.

At the time of writing, the only successful approximation of the program state coverage using also values from the program state's data is done surgically on targeted program points selected by a human [10]. Portions of the state space are manually annotated and the feedback function is modified to explore such space more thoroughly.

The automation of this process is a crucial topic in future research in this field.

In this thesis, we propose a new feedback for Fuzz Testing that takes into account not only Code Coverage, but also some interesting portions of the program states in a fully automated manner and without incurring path explosion.

We augment classic *Edge Coverage* — a type of code coverage based on edges in the *Control Flow Graph* — with information about "unusual" values in the program states observed in the incoming basic block.

We learn constraints between variables in basic blocks from executions traces of an input corpus, generally a corpus that is the output of a previous CGF fuzzing run, that describe how variables are related to each other and that holds for all the executions observed so far.

These constraints are mined *Program State Invariants* over basic blocks.

Execution-based invariants mining techniques, like the one that we use based on [33], suffer from the well-known *Coverage Problem* that means that the learned constraints may be only local properties of the observed corpus, and the violations of a learned invariant may not lead to a violation of the program specification. This, however, is not a problem for our purpose.

Local properties, if enough generic like the learned invariants tries to be, are still an interesting abstraction of the program state.

So we define a new feedback function that distinguishes the same edge with learned invariants in the incoming basic block that holds from the same edge with one or more learned invariant violated.

We develop a set of heuristic to produce invariants and techniques to effectively instrument programs with a low-performance overhead — a very important metric in fuzzing — and we implement them into a prototype called InvsCov on top of AFL++ [37].

In our evaluation, we show that a feedback that takes into account the program state abstraction can uncover more or different, software bugs than CGF.

1.1 Contributions

The key contributions of this thesis are:

- A new feedback that uses an abstraction of the program state from mined invariants;
- A prototype implementation based on LLVM and AFL++ called InvsCov;
- A systematization of the concepts behind Feedback-driven Fuzz Testing.

We plan to share the prototype as Free and Open Source Software.

1.2 Structure of the Thesis

In Chapter 2 we describe the basis of Software Testing and introduce various key concepts including the invariants. In Chapter 3 we generically describe Fuzz Testing and introduce a new abstract taxonomy for Feedback-driven Fuzz Testing. We discuss also some of the challenges of Feedback-driven Fuzz Testing. In Chapter 4 we introduce our methodology to abstract the program state's coverage. In Chapter 5 we present our prototype InvsCov and the technologies on which it is based. In Chapter 6 we evaluate the prototype in terms of efficiency and effectiveness. The thesis ends with the conclusions and the discussion of future directions in Chapter 7.

Chapter 2

Basics of Software Testing

Software Testing is the process that analyzes a *System Under Test* (SUT) to detect differences between existing and required conditions and to evaluate its features [2]. The most common embodiment of Software Testing is the process of finding software bugs.

Bugs cause the software to produce incorrect results or to behave unexpectedly. Bugs are a serious matter, they affect the everyday life of every person that depends on modern technology and, in the worst cases, cause even huge losses in terms of money [59] and human lives [61].

2.1 Correctness

In every stage of the system development, we can shape the existence of two entities: a *specification* and an *implementation* [58].

The development process converts the specification into the implementation. A high-quality implementation means to satisfy as much as possible the specification.

An implementation that completely matches the specification would provide the highest quality, but such equivalence is impossible to be stated for a Software Testing process, otherwise, with such a process it would also be possible to solve the Halting Problem [86].

So the *correctness* of a SUT cannot be expressed in terms of equivalence between specification and implementation.

In the following, we provide some needed definitions that match with [4].

Definition 1. *A* `failure` *is an externally visible deviation from the specification.*

Definition 2. *A* `fault` *(or* `bug`*) is portion of system state that leads to a failure. Note that if such a state exists but is never reached it does not cause a failure.*

Definition 3. *An* `error` *is a human error that causes the system to behave as not expected. Errors can cause faults.*

Given these concepts, we define the correctness as follows.

Definition 4. *A* `correct` *implementation of a specification does not contain faults.*

Correctness can be achieved by constructing a system without errors or by detecting and fixing all the faults.

In the first case, the absence of errors has to be proved formally.

2.2 Validation and Verification

The quality of a system in Software Testing is assessed by two processes: *Validation* and *Verification* [69].

Validation is the process that evaluates if the system really meets the needs for whom it was built. It is a subjective process that includes for instance user evaluations and prototyping. We will not discuss validation in this thesis.

Verification is the process, more objective than Validation, that evaluates if the implementation behaves according to the specification.

Verification processes aim to remove all the faults from the system, but this does not guarantee of course that the system has a value, this is stated by Validation.

Note that sometimes, for instance in the beta testing stages of the software development cycle, the two processes are combined.

2.3 Properties of Testing

For the sake of the verification of a system, a testing procedure may continue to add tests until all the faults are uncovered.

However, with constrained resources, this is impossible, so tests have to be prioritized.

Another property of testing is the context-dependent nature of the techniques. Testing the autopilot software of a plane requires far different methods than testing a toy like a Tamagochi.

Faults are often clustered, bugs are not uniformly distributed in a system. This, for instance, affects the prioritization of the tests.

Another very important property is, using Dijkstra's words, that software testing can be used to show the presence of bugs, but never to show their absence.

So, given these properties, we want a great diversity in our tests and so it is convenient to automatically generate tests.

2.4 Automation in Testing

Automation in Software Testing, most of the time, means sampling the input space of a SUT to generate testcases. The goal, according to [19], is to gain confidence about a certain degree of correctness or to find as many faults as possible.

There are also, however, unsuccessful techniques that try to find just one faulty input to prove incorrectness. The failure in proving incorrectness, of course, does not prove correctness, so we exclude these techniques from the treatment.

There are two types of sampling:

1. *Systematic*, in which the generation is informed by some artifacts from the SUT, like the specification;

2. *Random*, a uniform at random sampling of the input space that basically has no cost;

Given that we excluded techniques that find just a failing input, we can reduce Systematic sampling to a partitions-based approach.

Each partition is a subdomain of the input space and the inputs in each partition have common properties.

An effective type of partition strategy, as shown by [96], is the one that samples from *error-based* partitions. Each partition triggers an error or not. The testing strategy, given that is not known if a partition is associated with an error, samples each partition using a systematic approach. This is very effective but also hard to apply in the real world. Rather than that, several other automatic techniques are used in practice and we discuss some of them in Sec 2.5.

2.4.1 Efficiency Criteria

As the goal of Automated Testing is to gain a certain level of confidence about the correctness of the SUT or to uncover as much as bugs as possible, an efficiency criteria, as introduced in [19], must relate these goals with time.

In particular, in the first case the goal of an Automated Testing is to establish the level of confidence in a minimal amount of time, and, in the second case, to maximize the number of found bugs in a given time.

When evaluating Systematic techniques, an useful insight from [19] is that if we increase the effectiveness of the technique we have to increase the cost and so decrease the efficiency. This leads to a second important result that is when the SUT size is over a certain bound, Random Testing becomes more efficient than the evaluated Systematic approach.

2.5 Testing Techniques

We can divide testing techniques into different families. In this section, we will discuss some of them.

2.5.1 Specification-based Testing

To automatically generate a test several sources of information can be used. The first source of information for a Systematic approach is the specification.

Techniques based on this concept are called *Specification-based Testing* and do not require any knowledge of the structure of the program (i.e. programming language, size of the codebase, etc.).

The key idea is that several tests are derived from the specification and each test covers a *partition* of the System Under Test.

A common criteria, in term of partition-based testing, is to divide the input space into unique partitions that are unique in terms of exercised program behavior and in which is easy for an oracle to verify if such behavior, given one input, is correct.

2.5.2 Structural Testing

Another type of technique that aims to generate testcases using the code itself as a source of information is called *Structural Testing.*

A piece of typical information extracted directly from the code is the notion of *Coverage*, a notion that indicates how much code is exercised executing an input.

A prominent objective of this type of testing is to test all the code maximizing the coverage seen, but other criteria are supported as well and this affects how tests are generated.

Several types of coverage can be defined, following we discuss the most common classes:

- *Line Coverage* is probably the most straightforward type of coverage, related to lines fo code covered. However, this is a problematic coverage because it is affected by the coding style and programming language density;

- *Block Coverage* dope with the limitation of line coverage defining a more objective metric. Block refers to a block in the *Control Flow Graph (CFG)* [6], which is defined at a high level as the paths that can be traversed in the code. A *Basic Block* is an aggregation of adjacent code lines executed without a control flow change (i.e. no branches), a *Decision Block* is a block containing the predicate that affects a control flow change and an *Edge* is the connector between these blocks. A basic block has only one exit, a decision one has two exits, one of the condition is true, the other if it is false. Block coverage is simply when the testing technique aims to cover all the blocks in the CFG instead of all the lines in the code;

- *Edge Coverage* is always related to the CFG and it is used when block coverage is not enough in presence of complex branch conditions. Covering the edge coverage at 100% means that all the decision branch are exercised;

- *Path Coverage* is the coverage of all possible independent paths in the code. In term of the CFG, maximizing the path coverage means covering each possible path from each node to each other connected node;

2.5.3 Model-based Testing

A model of a system under test holds some properties and attributes of such a system in an abstract way. Using such abstraction tests can be generated.

A widely used type of model is the *Decision Tables*, tables that relate actions that the systems perform and conditions to take that action.

A prominent version of model-based testing is *State-machine based Testing.*

A state machine describes the system using states and transitions between these states.

To derive testcases from a state machine, like for structural testing, we can define some types of test coverage:

- State coverage: all the states have to be covered at least once;

- Transition coverage: each transition has to be covered at least once;

- Path coverage: exercise combination of transitions called paths;

The typical model-based testing workflow is bringing the system into different states and, after each transition, asserting that the system is in the expected new state.

2.5.4 Property-based Testing

In structural testing, we use information from the code to generate tests. In *Property-based Testing*, we use properties of the program to let the code itself to check the correctness.

The oracles that check if an execution of a test is related to a correct behavior are not anymore external but embedded in the code.

A common construct that developers employ to do that is the notion of *assertion*, a boolean expression inserted in a specific program point that, if false, reveals the presence of a bug.

Testcases can be then generated until an input that violates one of the assertions is found. *QuickCheck* [21] is one of the first tools developed that generates almost random inputs for the program under test in order to find testcases that violate one or more assertions.

Related to this type of testing is the software design pattern called *Design-by-contracts*. In this methodology, each caller component ensures that the preconditions to call a callee component are met. The main advantage is that errors are caught by the code itself avoiding propagation and the computation of incorrect results.

The checks that test the code during the execution are based on the concept of *Invariant*, a property that is always true at one or more particular program points, as described in [32].

Types of Invariants

Two widely known types of invariants are pre and post conditions of programs. Firstly introduced by [50], with the term *Hoare Triples* we denote the triple $\{P\}A\{Q\}$, where P is the pre-conditions that holds before the execution of the program A, and Q are the post-conditions that holds after the execution. Of course, A can denote also functions in a program or even single statements.

Example. Consider the following C function that pop an element from a stack data structure:

```
struct item* stack_pop(struct stack* stack) {
    struct item* item = stack->base;
    stack->base = item->next;
    stack->size--;
    return item;
}
```

In this case, we can define this Hoare Triple:

- P: `stack != NULL, stack->base != NULL, stack->size > 0;`
- A: the `stack_pop` function;
- Q: `stack->size == original(stack->size) -1,`
  `return == original(stack->base);`

You can easily see that if one of the preconditions is violated there is a fault (a Null Pointer Dereference or an Integer Overflow).

Another type of invariants are *Class Invariants* [49], that are strictly related to Object-oriented Programming. These invariants refer to an object or class and hold for the entire lifetime of such an object.

Loop invariants [38] [50] are predicates over the state of a loop that holds for every loop execution.

Also, other types of invariants were defined in the literature, for instance like the invariants based on concurrency constraints in [55].

Mining Invariants

Automatic invariants learning is a widely used process in verification, for instance for testcases generation [24] or memory errors detection [85].

A valid approach to learn invariants from the code of a SUT is static analysis, using techniques like *Symbolic Execution* [13], of which [92] is an example, or *Abstract Interpretation* [15], used in [40].

While invariants extracted using this type of analysis most of the time correct and without false positives, they are often overapproximation or simply the static analysis is not enough powerful to spot some invariants that are revealed only at runtime.

Opposite to that, many approaches like [46] [31] [76] uses information gathered during the execution, in a dynamic fashion.

One of the most popular tools to extract invariants from program traces is DAIKON [33], that uses a machine learning approach to learn from traces that are in an abstract form, allowing the tool to support many programming languages and runtimes like C, Java, C#, and many more.

The problem of these approaches is, unlike static approaches, that the extracted constraints are *likely invariants*, properties that hold at least for the observed executions. This, while allowing the analysis to reveal more invariants, leads to false positives with constraints that are only local properties of the observed executions. This problem is commonly called the *Coverage Problem* since, if the corpus of testcases used for learning does not cover the possible program state, at it is like to be in real applications, there may be still testcases with different coverage that violates the learned likely invariants.

Chapter 3

The Art of Fuzzing

In this chapter, we systematically review the concepts behind *Fuzz Testing*, a successful Random Testing family of techniques.

In particular, we focus on *Feedback-driven Fuzz Testing*, a technique that evaluates generated testcases using feedback from the *System Under Test (SUT).*

3.1 Generic Definitions

Definition 5. `Fuzz testing` *or* `Fuzzing`*, according to* [66]*, is the repeated execution of the Program Under Test (PUT) using inputs sampled from an input space and that stresses the PUT with unexpected inputs. We can generalize Fuzzing using SUT as subject instead of PUT.*

The usage of SUT is motivated by the fact that Fuzz Testing is nowadays extended to other domains, for instance, it can be used to test a web application composed of different distributed components or a set of programs communicating with each other like the IPC stack of a browser.

We simply call a program that implements a Fuzzing technique *Fuzzer*.

Definition 6. *A* `Violations Oracle` *is a process that determines if an execution of the SUT violates some requirements.*

An example of requirements is the correctness of the SUT or some performance requirements. We can also, for instance, assert that some code must not be reached if a specific configuration of the SUT is provided and use Fuzzing to try to find an input that executes that code under that configuration. Typically, many fuzzers look for crashes in a program, an easily observable type of failure.

Definition 7. *The* `Fuzzer State` *is the set of variables and artifacts (e.g. the instrumentation added to the SUT if any) that affects the behavior of the Fuzzer. They can evolve during the fuzzing process.*

Note that the fuzzer state is called configuration in [66], however, this is misleading while talking about tunable fuzzers like [37], so we use the term State instead.

A generic enough algorithm that can be used to define fuzz testing is 1.

Algorithm 1: Generic Fuzz testing

```
Result: The set V of testcases with violations
$V \leftarrow \emptyset$
$S \leftarrow \mathsf{Preprocess}(S)$
while $\mathit{Continue}(S)$ do
    $I \leftarrow \mathsf{InputGeneration}(S)$
    $S, V \leftarrow \mathsf{InputEvaluation}(I, O, S, V)$
return $V$
```

The abstract stages in the algorithms are defined accordingly to [66] but in a more generic flavor.

We define such stages as follows.

Definition 8. *The* `Preprocess` *stage is executed once before the fuzzing loop and modify the initial Fuzzer state.*

Definition 9. *The* `Continue` *stage decides, looking at the current state, if the fuzzer must stop or continue to search for violations.*

Definition 10. *The* `InputGeneration` *stage generate a testcase for the current run using the state.*

Some Fuzzers can use for instance previous testcases embedded in the state to generate a new testcase.

Definition 11. *The* `InputEvaluation` *stage is the stage responsible to feed the SUT with the newly generated input and use the violations oracle O to determine if this is a testcase that violates some of the requirements. It also updates the Fuzzer state with information useful in other stages.*

3.2 Fuzzers Classification

Until now we defined fuzzing in a generic way, unrelated to the actual System Under Test or to the specification.

We can go further in the definition of Fuzz Testing using classes based on how the Fuzzer generates the testcase and how much information about the SUT it needs.

Firstly, looking at the InputGeneration stage, we can define two types of generations:

- *Model-based* generation uses a model of the input format of the SUT embedded in the Fuzzer State to generate the testcase from scratch. It can be for instance a grammar specified in the initial Fuzzer state by a human (e.g. [51]), a mined model using learning techniques (e.g. [43]) or hardcoded generation rules in the algorithm (e.g. [97]);

- *Mutation-based* generation uses previous testcases called *corpus* that has to be provided in the initial state or saved in the state during the previous executions of InputEvaluation to generate the testcase modifying a testcase in the corpus;

Note that the mutational generation can use different strategies to mutate an input, even a model of the input format like model-based generation like in [7] [79]. The difference, in this case, is that the new testcase is not generated from scratch.

Orthogonally to the properties of InputGeneration, we can classify a fuzzer using the information that it needs from the actual SUT.

There are three common classes of fuzzers based on this criteria [41]:

- *Black-box* fuzzers does not need any insight from the SUT. The first attempts at fuzzing are of this type, which is closely related (if not even equal) to traditional Random Testing. Note that the lack of information from the actual implementation does not imply the lack of information about parts of the specification. For instance, black-box fuzzers like Peach [30] require a model of the input format to generate testcases;

- *White-box* fuzzers systematically inspect the state space of the SUT using internals information. White-box fuzzing can often overlap with Systematic Structural Testing. An example is SAGE [42] that tries to maximize code coverage using constraints gathered during the execution;

- *Grey-box* fuzzers stands in the middle of the two previous approaches. They collect minimal information from the SUT to better explore the input space while maintaining the performance overhead low. Traditionally, the information is collected during the SUT execution and it is code coverage, like in [99];

The distinction between these categories is often unclear, they are commonly used in the Security community but it is a debatable taxonomy, in this traction, we will avoid it and use the taxonomy from Software Testing when possible.

3.3 Feedback-driven Fuzzing

There is a strong empirical evidence [5] that fuzzing with code coverage as feedback from the SUT increases the efficiency in terms of the number of founds faults in a given time window.

Many fuzzers rely on code coverage (mostly use edge coverage [99] [63]) as a feedback, but the technique is not limited to this particular information and, as shown by [10] [75] [95], there can be many types of feedbacks (Sec. 3.3.4) that a fuzzer can use to better explore the input space.

Fuzzers typically employ evolutionary algorithms to process the collected information from the SUT. A general enough version of those algorithms is Alg. 2.

Algorithm 2: Basic Evolutionary Fuzz testing

```
Result: The set V of testcases with violations
$V \leftarrow \emptyset$
$S[Corpus] \leftarrow InitialCorpus$
$S[SUT] \leftarrow \mathsf{Instrument}(S[SUT])$                ▷ Preprocess
while $Continue(S)$ do
    $T \leftarrow \mathsf{PickTestcase}(S)$
    $N \leftarrow \mathsf{Calibrate}(T, S)$
    for $i \leftarrow 0$ to $N$ do
        $I \leftarrow \mathsf{InputMutation}(T)$        ▷ Mutation-based InputGeneration
        $S, V \leftarrow \mathsf{InputEvaluation}(I, O, S, V)$
return $V$
```

InputEvaluation in Alg. 2 is responsible to evolve the Corpus in the fuzzer state.

In Feedback-driven Fuzzing it executes the SUT with the generated input and, looking at the gathered feedback, if the execution is interesting it adds the testcase to the corpus. We can formalize Feedback-driven InputEvaluation as in Alg. 3.

IsInteresting is deeply connected with the feedback chosen, which has to maintain

Algorithm 3: InputEvaluation in Feedback-driven Fuzz testing

```
Data: The generated input I, the violations oracle O, the current fuzzer
      state S and the set V of testcases with violations
Result: The next fuzzer state S and the updated set V of testcases with
        violations
$Tr \leftarrow \mathsf{Execute}(I)$          ▷ Tr is the observed trace of the execution
if $\mathit{IsViolation}(O, Tr)$ then
    $V \leftarrow \mathsf{AddToSet}(V, I)$
if $\mathit{IsInteresting}(S, Tr)$ then
    $S \leftarrow \mathsf{EvolveCorpus}(S, I)$
return $S, V$
```

in the fuzzer state progress information. In the simple case of code coverage, in the state, there is the coverage seen so far and IsInteresting returns true only if there is a previously unseen coverage in Tr.

To better define Feedback-driven Fuzz Testing, we can systematically define all of its entities and how they are related to the abstract stages defined in 3.1. Note that these concepts are also abstract themself and we can use them to classify feedback-driven fuzzers.

3.3.1 Oracle

As defined in 3.1, a *Violation Oracle* inspects the execution of the SUT to decide if a testcase violates or not the expected requirements.

For Feedback-driven Fuzz Testing we add another property to the oracles, concerning the feedback: the ability to distinguish between violating inputs that are duplicates in terms of violated requirements or some other criteria.

Definition 12. *An* `Oracle` *is an entity that inspects the execution of the SUT to determine if it violates the given requirements. Besides, it decides if a violating testcase is worth to be added to the set of violating testcase V based on some criteria.*

We said that this new property is somewhat related to the feedback because, typically, fuzzers uses the feedback to distinguish between crashing inputs that probably trigger the same bug. For instance, AFL [99] use the coverage to distinguish crashes. If a new crash does not trigger new coverage in relation to the testcases already in V, it is discarded as a duplicate.

Other fuzzers like [91] use the hash of the callstack just after the crash.

Another common type of oracle is the one employed by differential fuzzing that tests if the outcome of an implementation is the same of the outcome of another considered correct [12].

3.3.2 Observation Channel

Definition 13. *An* `Observation Channel` *is an entity that provides information about the SUT execution to the fuzzer.*

The execution trace Tr in 3 is a possible outcome of an *Observation Channel.* It is used to get the information needed by the feedback.

An example of an observation channel is the shared memory of AFL. It is a shared bitmap between the target and the fuzzer that reports the coverage.

Note that the feedbacks uses the observation channels, but they are not limited to be used only for feedbacks [35].

3.3.3 Executor

The concept of executing the SUT is not always the same. For instance, for in-memory fuzzers like LibFuzzer [63] the SUT is a harness function, for hypervisor-based fuzzers like KAFL [88] instead the SUT can be the entire operating system.

All fuzzers, however, needs the same primitives to execute a SUT.

Definition 14. *An* `Executor` *is an entity with a set of violation oracles 3.3.1, a set of observation channels 3.3.2, a function that allows instructing the SUT about the input to test, and a function to run the SUT.*

Placing an input in the SUT can be a very different task depending on the type of executor. LibFuzzer just place it as arguments of the harness function, while other fuzzers can write to the program standard input or in a specific memory region when executing the target inside an emulator or a similar controlled environment.

3.3.4 Feedback

As the technique is Feedback-driven, the concept of *Feedback* is quite important.

The fuzzer must interpret the information retrieved using the observation channels and relate it to the fuzzer state.

Definition 15. *A* `Feedback` *is an entity that defines how to interpret the gathered information from observation channels and how to evolve the corresponding fuzzer state, specifically how to evolve the testcases corpus based on that information.*

For instance, given an observation channel that reports the size of memory allocations, a feedback that aims to maximize these allocations size to spot out-of-memory bugs can be defined with the following sentence:

Given a map M from the observation channel, each entry corresponds to a single program point and it is a 64-bit unsigned integer. The fuzzer state must maintain an accumulation map A that, for each registered allocation program point, maintains the maximum size of the allocation seen so far. A testcase is added to the corpus (IsInteresting) if when updating A with the corresponding M, at least one entry is updated.

In literature, some fuzzers make use of feedbacks that does not simply aim to maximize code coverage, like PerfFuzz [60] that maximizes the execution counts of all the program locations to spot performance issues and FuzzFactory [75] that implements different feedback functions based on maps.

Note that many fuzzers use a map as observation channel and a reduce function to evaluate if the collected information is interesting, but the notion of feedback is not limited to this particular embodiment.

3.3.5 Input

Until this point, we did not specifically define what is an input for the SUT because it is an abstract concept. It is, in general, a sample from the *Input Space*, all the possible data that the SUT can take from an external source and that affects its behavior.

In the straightforward case, the input is a simple file or buffer passed to a program, but it can be also, for instance, a sequence of actions [80] or even different values read in different program points independently [93].

Definition 16. *An* `Input` *entity defines one possible sample from the Input Space and can hold properties about the input itself, the relation between the input and the SUT, or the input and the specification.*

Note that the description of how the SUT consumes the input is provided by the Executor.

Input metadata

We refer to the properties that the Input can hold as *Input metadata*.

An example of metadata, that relates the input to the specification, is the virtual structure.

Fuzzers such as AFLSmart [79] or Nautilus [7] maintains a tree representing the Abstract Syntax Tree of the input when parsed using an *Input Format Specification* provided by the user. That information comes from the specification of the SUT (e.g. we know that the program under test process PNG files) and it is used in the mutator 3.3.7 to perform structure-aware mutations.

Another example of metadata is the tags extracted by Weizz [35] from the SUT using dynamic analysis. Like in the previous example, this metadata is used for mutation, but it relates the input with the SUT and not with the specification, because they are extracted using an observation channel.

A third, and naive, example of metadata that related the input with the SUT is the execution time, and one that is a property of the input itself is the size in bytes if it is a buffer. This kind of metadata is often used in AFL-like fuzzers to prioritize simpler inputs.

Scheduling inputs

The outcome of Calibrate in evolutionary fuzzing is affected by the current Input T extracted from the corpus.

Calibrate controls how many fuzzing iterations have to be done mutating a testcase. It can employ algorithms based on the input metadata, for instance, the execution time.

These algorithms that schedule how many iterations are assigned to a testcase are commonly referred to as *Power schedules.* Several works addressed this problem like [20] [18], using, for instance, the triggered coverage as input metadata. An interesting insight from these works is that is convenient for some fuzzers to give more iterations to testcases that cover program points that are rarely stressed.

3.3.6 Corpus

The Corpus is a set of inputs in the fuzzer state. In Feedback-driven Fuzzing, it is necessarily connected to the Feedback.

Definition 17. *A* `Corpus` *is an entity that collects inputs that are interesting for one or more feedbacks, and defines how they are related to each other and how to feed the fuzzer with those inputs when requested.*

The Corpus is implemented as a data structure containing inputs, but it can differ a lot for each fuzzer.

For instance, AFL uses a queue to store inputs, but other fuzzers use just a simple container that feeds the fuzzer using a random selection algorithm.

The Corpus is also not unique. There can be a corpus that contains inputs interesting for just one feedback.

Corpus transitions

The Corpus can use inputs metadata to relate the inputs to each other and schedule how to serve testcases to the fuzzer when it needs them for a fuzzing run.

We can view the Corpus as an evolving entity not only when adding a new testcase, but also when the fuzzer requests the next testcase to fuzz.

We call this second type of evolution from a request to another *Corpus transition*.

When a Corpus randomly selects the next testcase, the transition is the simplest. When it is a queue, the transition is the `get` operation of the queue.

More complex fuzzers employ algorithms to select smartly the next testcase. For instance, AFL uses the coverage triggered by each input to create a minimized subset of the Corpus that covers the entire coverage seen so far. With a high probability, only inputs in this subset, that is periodically updated, are fuzzed.

3.3.7 Mutator

Definition 18. *A* `Mutator` *is an entity that takes one or more inputs and generates a new derived one.*

A mutator can both modify the input and the related metadata. Some mutators work just on the metadata and then the change is replaced to the testcase.

The concept of mutator is deeply linked with the definition of the input for the SUT, typically for each type of input, there are specialized mutators.

Scheduling mutations

A mutator most times can apply more than a single type of mutation on the input. Consider a generic mutator for a byte stream, bit flip is one of the possible mutations but not the single one, there is also, for instance, the random replacement of a byte of the copy of a chunk.

When a mutator has a collection of mutations, what mutations have to be used for the specific input is a problem addressed scheduling the mutations.

Naively, for most fuzzers, the number of mutations is a random bounded number and the sequence of mutations is randomly chosen too. In more complex approaches, a scheduling algorithm is chosen.

For instance, MOpt uses particle swarm optimization to select mutations based on their effectiveness in finding new interesting input in past iterations of the fuzzer.

3.3.8 Generator

Definition 19. *A* `Generator` *is an entity that generates a new input from scratch possibly using some parameters.*

Opposed to input creation by mutation, there is input creation by generation. Feedback-driven Fuzzing is most of the time related to Evolutionary Fuzz Testing that, of course, needs a mutator to evolve the corpus. There are, however, situations in which generators are used in this kind of testing.

One is when the generator is invoked by a mutator, like when mutating a virtual structure of the input. Consider a mutator that operates on the Abstract Syntax Tree, it can randomly replace a subtree with a new one generated from scratch as a mutation. In this case, generation is a mutation.

Another situation is when the SUT itself asks for the input the fuzzer in different stages (e.g. a stateful network protocol when a request-response sequence is considered a single input), and the entire input that was initially created by mutation was already given to the SUT. In this case, the fuzzer has to generate an extension of the input previously generated by mutation and can do that using a generator.

A third situation, less explored, is using Feedback-driven Fuzzing not to evolve a corpus of testcases, but a corpus of parameters for the generator. Consider a grammar-based generator, a parameter that affects the generations is, for instance, the probability to choose a terminal node or go deeper in the grammar and continue exploring a nonterminal. These parameters can potentially be tuned using the feedback.

3.3.9 Stage

Definition 20. *A* `Stage` *is an entity that operates some actions on a single input.*

This definition of Stage is very abstract, we used it because it is just a proxy used as a building block of the fuzzing algorithm.

A mutational stage, given an input of the corpus, applies a mutator and executes the generated input one or more time. How many times this has to be done can be scheduled, AFL for instance use a performance score of the input to choose how many times the havoc mutator should be invoked. This can depends also on other parameters, for instance, the length of the input if we want to just apply a sequential bitflip, or be a fixed value.

A stage can be also an analysis stage, for instance, the *colorization* stage of RedQueen that aims to introduce more entropy in a testcase or the *trimming* stage of AFL that aims to reduce the size of a testcase. A possible stage can also, for instance, execute the SUT with particular instrumentation to extract some input metadata, like in Weizz.

3.4 Challenges

The latest state-of-the-art research in Fuzz Testing tries to address some of the challenges that make fuzzing less efficient.

3.4.1 Roadblocks

The most straightforward limitation, when dealing with coverage as feedback, is the code roadblocks for the fuzzer. Multi-byte comparisons are one type of roadblock because, given a generic byte stream mutator, it is nearly impossible to guess the exact value of the other operand of the comparison to flip the branch.

Example.
Consider the following C code snippet:

```c
void foo(int x) {
  if (x == 0xbabdcafe)
    bug();
}
```

If x is the input provided by the fuzzer, the probability that the branch is flipped guessing that x should be 0xbabdcafe is almost 0.

In literature, this issue is addressed using a feedback that track the progress of the comparison [3] [62], with concolic execution combined with fuzzing [98] [81] or with techniques that extract the comparison values and try to replace patterns in the input [84] [11].

Other common roadblocks are the checksums. They are mostly used to protect chunks of binary formats against corruption, the probability that a generic fuzzer mutates the protected bytes and, at the same time, restore the checksum field validity is almost 0.

This problem can be addressed using a specific mutator for the binary format or with code transformation that patch the program removing the checksum checks [11] [78] [35].

3.4.2 Invalid inputs

Another challenge of fuzzers with generic mutators is the high rate of generated invalid inputs.

When the mutator likely breaks the validity of the input, the fuzzer stresses most of the times the code related to parsing, but not deeper code.

In order to effectively fuzz deep paths, the fuzzer must produce valid inputs, and this can be achieved using a model of the input format to guide the mutator, like in [79] [7].

Some techniques tried to approximate that automatically, without a human written model [16] [35] [68].

Another approach is to constrain the mutator to not touch the portion of the input that leads to the deep path and even constraint the possible values that an input field can take using, for instance, constraints collected using concolic tracing [52].

3.4.3 Faults without Failures

Most oracles in fuzzers use failures to know if a testcase violates the requirements, in particular crashes, but sometimes a fault does not directly trigger a failure.

To catch these kinds of bugs, the SUT is often instrumented with additional tripwires to catch silent faults. For instance, a one-byte overflow in read on the heap will unlikely trigger a crash in a C program. To handle this situation, source-based fuzzers offer the possibility to instrument the programs with sanitizers such as AddressSanitizer [89]. Others make use of binary-only tripwires to uncover silent corruptions [70], inserted both dinamically, like the QEMU-based sanitization available for AFL++ [36], or statically [29] [39].

These sanitizers however cannot catch some pure logic bugs, and fuzzing to uncover this kind of bugs automatically (e.g. without putting manually assertions in the code) is an open field of research.

3.4.4 State Tracking

The code coverage, or some extension of it like comparisons feedback, is often not enough to explore the state space of a SUT. For instance, the code that handles a specific type of chunk in a binary format is considered always the same in terms of coverage, even when the previously processed chunk is different. The interleaving of different chunks is still an interesting property of those programs and often bugs are related to it, but most fuzzers cannot get feedback from it because code coverage does not notice it.

The current method to overcome this challenge is to manually select some state variables and get feedback from them manually [10].

The technique proposed in this thesis tries to address this particular challenge automatically.

3.4.5 Path Explosion

When the sensitivity of a feedback increases, for instance when adding feedback about the progress of comparisons to edge coverage or when using context-sensitive coverage, the fuzzer may save as interesting too many testcases. These testcases will be never processed all, and the fuzzer saturates. An example of feedback that easily leads to path explosion, as described in [95], is memory coverage, a feedback that considers a testcase as interesting if the execution of the SUT uses a previously unseen zone of memory.

Path coverage too easily leads to path explosion, and it is one of the most prominent problems of techniques that use it like Symbolic Execution [13].

3.4.6 Scaling Implementations

A current limitation of available fuzzers implementation is scaling on multiple CPU cores. System calls used by the fuzzer for various tasks are often not designed for scaling, they use expensive locks in the kernel or performs unneeded slow tasks for our purpose. The usage of Inter-Process Communications primitives offered by the OS to communicate between the fuzzer and the target instrumented program is an example.

A possible solution to this problem is embedding the fuzzer in the target application itself, like LibFuzzer does, and use a custom kernel extension or hypervisor to snapshot the program state in case of code that cannot be fuzzed stateless. Mocking the syscalls performed by the target helps too, usually, this task is performed when using an emulator to instrument the program.

Besides that, note that an exponential increase of the cores, as shown in [17], increases only linearly the ability to find new coverage or new bugs.

3.4.7 Hard Targets

The problem of instrumenting, executing, and feeding a SUT with the produced testcases is not trivial in practice.

While traditional tools like AFL instruments programs that take their input as a file, many applications and systems are not designed to behave in that way.

On the other hand, many programs can be adapted to consume a buffer produced by the fuzzer, like when using a LibFuzzer harness, and others can be instrumented to request pieces of input when needed to the fuzzer instead of using a single buffer [93], but there are targets that consume their inputs in other ways that are either not easy to handle or for which even execution or instrumentation is hard.

In the first case, a SUT of this kind is a complex system like a kernel or a multi-process environment like IPC stacks. For instance, SyzKaller [94] provides inputs to the kernel using a sequence of syscall invocations and fuzzing using a hypervisor with incremental snapshots [8] can be used to test multiprocess environments that share messages.

In the second case, the SUT is typically an embedded system. Often the execution of firmware is possible only on specific hardware, which makes instrumentation and hardening to catch silent faults impossible [70]. To address this problem, the code can be executed on an emulated hardware [100] [77], with a high cost in terms of development effort, or re-hosted [22] [65], a technique that transfers the context from the target to an emulator and forwards the interactions with the hardware back to the device only when needed.

3.5 Evaluation Criteria

The evaluation of fuzzing techniques is a matter currently under debate in the community.

A recent work [54] analyzed some papers on fuzzing and stated what should and should not be done to compare fuzzers. The number of testcases or reported crashing inputs is not an evaluation metric, and using them is a very bad practice. This work states that an effective metric is the number of triaged bugs triggered by each fuzzer or the coverage over time as a good proxy.

However, this approach is limited and too general when we want to evaluate the properties of fuzzers. A fuzzer that triggers less but unique bugs is good and, in with the same spirit, a fuzzer that covers unique coverage is good too. Recently, this metric of counting unique code blocks covered was introduced in FuzzBench [53].

Other metrics that are useful to evaluate fuzzers are which fuzzer reach a certain coverage in less time [34] and the number of hits of each block for some randomly sampled generated input, in order to evaluate the ability of a mutator to stress deep paths [9].

Chapter 4

Methodology

Code coverage as feedback for Feedback-driven Fuzz Testing is a successful proxy to approximate the program state during the exploration of the paths performed by the fuzzer.

Edge Coverage-based Fuzzing found thousand and thousand of bugs in complex applications in the lastest years [5], however, it suffers in exploring program states that lead to bugs but that are not directly related to code coverage.

A naive solution may be a Fuzzing algorithm that uses code and variables or memory values as feedback, but this quickly leads to path explosion, as shown by [95].

In this chapter, we describe or technique that tries to cope with this problem augmenting the feedback given by classical Edge Coverage-based Fuzzing using *basic blocks invariants* violations to approximate the program state coverage.

4.1 Definitions

In this section, we introduce some needed definitions for the rest of the chapter, in part according to [27].

Definition 21. *A* `Program` *is a sequence of instructions.*

In this dissertation, the SUT is a program.

Definition 22. *The* `Memory State` *is a function $M(a) \rightarrow v$ that associates a* `Memory Address` *a to a value.*

Definition 23. *The* `Static Single Assignment form (SSA)` [87] *is a type of intermediate representation (IR) in compiler theory. SSA requires that every variable is defined before each use and assigned only once.*

Without loss of generality, every program can be converted to SSA [25].

Example. Consider this simple C function that computes the maximum of two integers:

```
int max(int x, int y) {

  int m;
  if (x > y)
    m = x;
  else
    m = y;
  return m;

}
```

We can convert this simple program to SSA avoiding assigning two times the variable `m`. At first glance, this seems impossible because its value depends on the control flow, but we can use the Φ operator of SSA. This operator defines a new variable choosing between two possible values depending on the control flow.
For our simple example, we can define three different SSA variables that represent `m` in the two blocks of the if statement and in the terminal block that ends the function.
The translation to an SSA pseudocode is then the following:

```
Function max($x$, $y$)
    if $x > y$ then
        $m_1 \leftarrow x$
    else
        $m_2 \leftarrow y$
    end
    $m_3 \leftarrow \Phi(m_1, m_2)$
    return $m_3$
```

Definition 24. *The* `Load` *instruction $v := M(a)$ assign to a variable the value associated with the address a, that is a variable itself.*

Definition 25. *The* `Store` *instruction $M' := Store(M, a, v)$ alter the codomain of a memory state changing the value corresponding to a with v.*

Definition 26. *A* `Basic Block` [1] *is a straight-line sequence of instructions in the program with only one entry point and only one exit.*

Definition 27. *The* `Control Flow` *is the order in which each individual basic block is executed and evaluated. A* `Control Flow instruction` *is an instruction that changes that order.*

Definition 28. *The set of the* `Live Variables` *at a basic block is the set of variables that are used in the block or possibly used in any block reachable from it later in the execution.*

This set can be calculated using a backward analysis called *Liveness Analysis* [27]. Note that as in SSA a variable is used only once, redefinition is not an issue for our definition of liveness.

Definition 29. *A* `Program State` *is the triple* (l, V, M)*, in which* l *is the program point of the next instruction to be executed,* V *is the set of the live variables and* M *is the current memory state.*

In our simplified model of the computation in a program, this definition of program state alone is enough to describe the data in the program.

Note that, if we have to observe a program state during the execution of a program, we have to do it after the execution of the current instruction, because the next instruction l is determined only after the evaluation of the instruction in case of control transfer instructions.

4.2 The Basic Block State

Definition 30. *The* `State of a basic block` *is the set of all the SSA variables values used in the block.*

Such states can be easily observed after the execution of the basic block because SSA variables are assigned only once and so all the used values are still in the variables.

We also define a family of functions that describe basic blocks.

Definition 31. *Given a basic block* X*, the function* $BB_X(V_I, M_I) \to (V_O, M_O)$ *is the function that takes the set of live SSA variables* V_I *and the memory state* M_I *as they are before the execution of* X*.* BB_X *returns a new set* V_O *of SSA variables that are the variables used in* X *(note that* $V_I \cap V_O$ *is, in general, not empty) and the memory state* M_O *that is the memory state of the program after the execution of* X*.*

Following this notation, V_O is thus the basic block state.

Always using the properties of SSA, we can derive this property of the basic block state:

Theorem 1. *The side effect in memory of a basic block, $M_O \setminus M_I$, is a subset of the basic block state V_O.*

Proof. In an SSA IR, only the store instruction can affect the memory state. A store instruction s takes in input the memory location and the value v that has to be stored and so if s is part of a basic block, v is a variable in the basic block state. $\square$

4.3 Program State Abstraction

Definition 32. *Given an observed execution of a program, a* `Program Trace` *can be described as a chain of BB_X functions that take as input the output of the previous function in the chain.*

The order of the functions in the chain is given by the order of the executions of the basic blocks in the observed execution of the program, which is the Control Flow.

We can use the program trace to observe the transitions between program states, and, using the following theorem, we can do it efficiently at the end of each block instead of for each instruction.

Theorem 2. *Given a program trace, the observation of each basic block state from the outcome of each BB_X function provides the same information about changes in the program state of observing the values during each program state transition.*

Proof. A transition in the program state causes a change in:

1. obviously, the program point;
2. in the set of live variables, if a new variable is defined or another is not live anymore;
3. in the memory state, if the current instruction is a store;

We know, from Theorem 1, that the new values in the memory store are also tracked in the basic block state.

Given that the control flow of a basic block is unique, if we observe the basic block state after the execution of a block, we get that:

1. each instruction was necessarily executed, so we know all the program points in the transitions;

2. all the live variables for each instruction are still in the basic block state because, by definition, the liveness as a basic block granularity, and the newly defined variables are tracked because we observe them at the end of the block, so after all the definitions;

3. the changes in the memory states are in the basic block state;

So, observing the changes in terms of values in the program states inside a basic block at the end of the execution of such block is the same as doing it for each instruction. □

Note that for our theoretical model we are assuming a program without exceptional control flow, as usually these paths are not interesting to fuzz because related to errors and our treatment is simplified.

For each block, this information is a valid approximation of observing each individual program state.

Obviously, in a real-world implementation, logging a program state is not feasible even for a small number of executed instructions due to the occupied space. Logging the changes is a good trade-off and, as shown by Theorem 2, we can do it at the end of each block.

Now consider the basic block state as a space on the SSA variables. We can divide such space into subspaces using relations between the SSA variables.

These relations define hyperplanes in such space, and so also subspaces are implicitly defined by these hyperplanes.

Example. Consider a basic block state with two variables, x and y. If we have, for instance, the relations $y > x - 8$ and $x < 100$ we have that the following four subspaces are defined: $y > x - 8 \land x < 100$, $y > x - 8 \land x >= 100$, $y <= x - 8 \land x < 100$, $y <= x - 8 \land x >= 100$. If we violate one of the relations, we are in another subspace than when we do not violate them. If we violate both, there is another subspace.

A main idea is that if these relations describe coherently different partitions of possible values of the variables (e.g. there is a bug if $x\%2 = 0$) we can use the produced subspaces as an abstraction of the program state in the basic block.

4.4 Mining Subspaces

The relations between variables in a basic block state that delimitate the subspaces have to be chosen carefully in order to have a meaningful division of state space.

As our goal is to find bugs, an effective definition of such relations can be as *Basic Block Invariants*. This type of invariants, used in works like [23] [40], are relations that theoretically always hold in a basic block and describe all the possible values of a variable.

When one or more of such relations are violated, we are in a subspace that is related to an incorrect program state reached in this basic block.

As the SSA form guarantees us that the input variables of a block are not modified, we can avoid checking the pre-conditions of a Hoare Triple over the basic block because the same invariants are also present in the post-conditions, that use the basic block state. Note that in the basic block state the memory information is only related to the changes made in the current block, so implicit constraints between variables and memory values are missing.

This could be an effective technique to detect bugs, but mining these relations is hard.

As described in 2.5.4, some approaches try to learn the invariants with static analysis techniques, others try to learn the invariants from many executions traces of the program under test.

The first approach may miss invariants and the second may generate relations in which the violations are not related to a real bug, but is a local violation regards the learned data from the corpus of the execution traces, a likely invariant.

These over-approximated invariants are however interesting even if the violations maybe not be related to a bug. In the abstraction of the program state, a local property is still interesting and the definition of subspaces based on this kind of invariants is still a valid approximation.

Thus, we can exploit the coverage problem to learn relations about common states of the variables and define the subspaces as spaces in which the variables assume “unusual” values that may or may not be related to a bug.

4.5 An Invariants-based Coverage

Given the relations extracted using an execution-based invariants mining technique, we can define a new type of feedback for Fuzz Testing that is based on the abstract program state coverage defined by the subspaces of the basic block state.

The control flow information is, like in traditional Coverage-guided Fuzzing, approximated by the observed edges in the Control Flow Graph. This information is augmented with the subspace in which the variables of the incoming block are located.

So, a fuzzer saves an input not only when a new edge previously unseen is executed, but also when the program explores a new subspace of the incoming basic block state for that edge.

Example.
Consider the basic block from the previous example in Sec. 4.2. There are four possible spaces defined by two invariants.
Assuming that the block can generate N possible branches, it can produce $N * 4$ different feedback items for the fuzzer.
However, the observation of all the $N * 4$ cases is an extreme case in which all the likely invariants can be violated at the same time. In practice, some invariants may never be violated when another is violated too, or never violated at all.

To check if a particular state violates the learned invariants for a block, this technique has to emit these checks at the end of the blocks using code generation.

During the execution, the boolean information about the violation of a single invariant is then propagated to the next executed block to report the edge information augmented with the position of the observed values in the defined subspaces of the incoming basic block.

4.6 Pruning Invariants

Variables in the basic block state are, however, not always related to each other and this can produce useless likely invariants.

Besides, invariants that are impossible to violate, even in case of a fault, are not relevant for our technique.

To cope with these two problems, which pollute our coverage and increase the number of checks that must be generated — so increasing complexity and decreasing the execution speed of the program under test — we devise some optimizations for the invariants miner.

On the side of invariants checking, after the miner extracts the likely invariants, we can avoid checking for duplicate invariants to speed up the execution.

4.6.1 Comparability Calculation

Algorithm 4: Comparability set computation

```
Data: The IR function F containing all the IR instructions in that function
Result: The function $C\colon Instructions \longrightarrow Comparability$ that relates an
        instruction I to a comparability id
$C\colon \mathsf{GetAllValues}(F) \longrightarrow \{\epsilon\}$
for $I$ in $\mathit{GetInstructions}(F)$ do
    if $\mathit{IsUnary}(I)$ then
        $C, Id \leftarrow \mathsf{MergeComparability}(C, Id, I, \mathsf{GetOperand}(I, 1))$
    else if $\mathit{IsCast}(I)$ then
        $C, Id \leftarrow \mathsf{MergeComparability}(C, Id, I, \mathsf{GetOperand}(I, 1))$
    else if $\mathit{IsBinary}(I)$ then
        $C, Id \leftarrow \mathsf{MergeComparability}(C, Id, I, \mathsf{GetOperand}(I, 1))$
        $C, Id \leftarrow \mathsf{MergeComparability}(C, Id, I, \mathsf{GetOperand}(I, 2))$
    else if $\mathit{IsGEP}(I)$ then
        $C, Id \leftarrow \mathsf{MergeComparability}(C, Id, I, \mathsf{GetPointerOperand}(I))$
        $O_1 \leftarrow \mathsf{GetIndexOperand}(I, 1)$
        for $O$ in $\mathit{GetIndexOperands}(I) \setminus O_1$ do
            $C, Id \leftarrow \mathsf{MergeComparability}(C, Id, O_1, O)$
    else if $\mathit{IsLoad}(I)$ then
        $C(I) \leftarrow Id$
        $Id \leftarrow Id + 1$
return $C$
```

Firstly, we try to create independent sets of related variables in a IR function.

Variables that are not directly related but are related to a common third variable are in the same set. These sets are called *Comparability* sets and can be encoded using a function $C\colon Instructions \longrightarrow Comparability \cup \{\epsilon\}$ that relates each instruction in the function to a comparability set, identified by an ID in this case, or to default comparability ϵ, that represents the comparability with all the other variables.

Algorithms 4 and 5 describe how we compute C.

Each variable is initially related to ϵ, then, the list of the instructions is walked and if the current instruction is of a certain kind (e.g. a binary operator) each

Algorithm 5: MergeComparability auxiliary algorithm

```
Data: The function $C\colon \mathit{Instructions} \longrightarrow \mathit{Comparability} \cup \{\epsilon\}$, the
      progressive counter $Id$, the IR values $V_1$ and $V_2$
Result: The function $C\colon \mathit{Instructions} \longrightarrow \mathit{Comparability} \cup \{\epsilon\}$ and the
        progressive counter $Id$
if $C(V_1) \neq \epsilon \wedge C(V_2) = \epsilon$ then
    $C(V_2) \leftarrow C(V_1)$
else if $C(V_1) = \epsilon \wedge C(V_2) \neq \epsilon$ then
    $C(V_1) \leftarrow C(V_2)$
else if $C(V_1) \neq \epsilon \wedge C(V_2) \neq \epsilon$ then
    $C(V_1) \leftarrow Id$
    $C(V_2) \leftarrow Id$
    $Id \leftarrow Id + 1$
else
    for $V$ in $\mathsf{Domain}(C)$ do
        if $C(V) = C(V_2)$ then
            $C(V) \leftarrow C(V_1)$
return $C, Id$
```

variable is marked as related to the other. For instance, if the instruction is an addition, the comparability set of the result is merged with the sets of the two operands. The first time that a variable is hit, a new comparability id is assigned instead of ϵ. Variables that are used in instructions that are not in the cases shown in 4 maintain the ϵ comparability, an over-approximation needed to not lose interesting invariants.

The kinds of instructions that are considered in Alg. 4 are the unary instructions, operators with a single operand, binary instructions, with two operands, cast instructions, that convert a value to another with a different type, load instructions, and GEP instructions, that compute an address given a pointer and a set of indexes.

4.6.2 Inviolable Invariants

As our technique is based on violations of likely invariants, we want to avoid the generation of invariants that are always inviolable, also in the case of a fault. Learning basic invariants from the actual code of the program leads to such types of invariants.

Example.
Consider a **unsigned int** variable in C. It will be mapped to many different IR variables, but all of them will never be negative. An inviolable invariant is so that these variables are always greater than or equal to 0. This is a useless check for the invariants' coverage, it will be never violated.

Value Range Analysis [47] is the technique that, if used with a conservative approach, allows us to define bounds to integer variables that always hold.

Consider the definition of a constraint variable Y in one of the following ways:

- $Y = [a, b]$
- $Y = \mathsf{Merge}(X_1, X_2)$
- $Y = X_1 + X_2$
- $Y = X_1 * X_2$
- $Y = a * X + b$
- $Y = X \sqcap [a, b]$

The Merge operator merges two variable names into one [26], $\sqcap$ is the range intersection.

These definitions can be easily extracted from a program in SSA form.

The Range Analysis objective, as explained in [83], is to solve a constraint system with variables in the same form of Y and associate each variable to an integer range (with $+\infty$ and $-\infty$).

So, with this technique applied to SSA variables used to produce invariants, we can extract over-approximated ranges for each variable and exclude these invariants from the learning output, as they always hold.

4.6.3 Deduplicate Invariants

Basic blocks defined in terms of a sequence of instructions in a program are, like generic basic blocks defined in 2.5.2, nodes in the *Control Flow Graph (CFG).*

We can define dominance relations between blocks in the CFG as in [82].

Definition 33. *A basic block A* `dominates` *a basic block B if every path in the CFG from the root to B must go through A.*

Related to this definition, there are other definitions:

Definition 34. *A basic block A* `strictly dominates` *a basic block B if A dominates B and $A \neq B$.*

Definition 35. *A basic block A* `immediately dominates` *a basic block B if A strictly dominates B but does not strictly dominate any other block that strictly dominates B too.*

The immediate dominator is unique and every block (except the root one) has one.

Definition 36. *The* `Dominator Tree` *is a tree defined with the basic blocks as nodes and the edges as the immediately dominates relations.*

As the immediately dominates relation is unique for the dominated block, the dominator tree is a tree in which a block can immediately dominate multiple blocks but can be immediately dominated only by one.

From a node, we can go backward in the dominator tree to find all the nodes that strictly dominate such nodes.

We can make use of such definitions to develop a method to prune duplicate invariants between blocks.

After the likely invariants phase, the produced checks have to be placed in the code and the outcome of each check is the identifier of the invariant if violated. If two IR variables are used in two blocks, the blocks will likely share an invariant.

With the dominator tree of a function, we can optimize this phase and avoid the insertion of redundant checks.

The actual check is emitted only for the top-level (the nearest to root) dominator that shares the invariant, and the outcome is propagated to the dominated blocks without the need to execute again the check.

4.7 Corpus selection

We rely on a mining approach based on execution traces and this needs several test-cases to generate different traces. Like for previous evolutionary fuzzing techniques, the choice of the initial corpus is critical.

Unlike traditional CGF, in which indeed the input corpus affects by a lot the performance of the fuzzer, an unwise choice of the initial corpus for our technique can produce biased results, not just degrade performance.

For instance, it is a common practice to download many files of a given file format when testing a parser, but those files are almost all valid files. If we learn likely invariants from the execution of a similar corpus, we will bias our invariants on the validity of the file format and, in some cases, this can be a mistake because we miss interesting partitions of the basic block states related to invalid inputs.

As our technique aims to improve the ability of a fuzzer to better explore the code regions that were already reached executing the initial corpus, an interesting choice is to mine invariants over the corpus of another fuzzer.

This is interesting because we have nowadays fuzzers that reach very good coverage in a reasonable time (e.g. [11] [81]).

A derived problem is then when should we stop the first fuzzer and apply our technique?

We can randomly select a time window, or wait that the fuzzer saturates in coverage [45].

The saturation is a known problem in fuzzing, and doing the latter can help to cope with this problem, but sometimes a fuzzer can get stuck in the opposite way: it will continue to find new coverage and fail to deeply explore a single code region. In this case, a random timeout is reasonable, maybe combined with the partial instrumentation of some selected regions of the programs to avoid this problem.

4.8 Discussion

With the proposed feedback in this chapter, an edge can be registered as different feedback values and introduce novelty more than a single time. To do that, we divide the space described by the IR values used in the incoming basic block using functions over these values. These functions must meaningfully represent interesting properties of such space, and so we used learned likely invariants over the basic block.

We devised a set of algorithms to reduce the number of generated invariants that are redundant, that do not produce feedback or that relate unrelated variables.

The proposed technique aims to augment the feedback to use not only information about the control flow but also about the entire program state to better explore the possible states of the SUT during Fuzz Testing.

Chapter 5

Implementation

In this chapter, we provide an overview of the technologies used to implement our technique, LLVM [56], Daikon [33], and AFL++ [37], the general architecture of our implementation and some details about it.

5.1 The Low Level Virtual Machine Infrastructure

The *Low Level Virtual Machine Infrastructure* (LLVM) [56] is a compiler and toolchain infrastructure designed for easy development of compiler frontends for programming languages and backends for instruction set architectures.

The core of LLVM is its intermediate representation (IR) that is frontend-agnostic, portable, and SSA-compliant. This IR allows compiler architects to implement optimizations and code analysis passes at many stages of the compilation pipeline in a completely language-independent flavor.

The generation of the machine code in LLVM can happen at compile-time, for each module, at link-time, enabling a wide range of aggressive inter-procedural optimization like arguments promotion [57], or even at run-time, using the LLVM just-in-time engine.

The supported backends at the time of writing are almost all the most used architectures, including X86, PowerPC, ARM, and SPARC, but also less known ones like Hexagon or WebAssembly.

Since our techniques are designed to work at the IR level, we can build an implementation that is architecture-independent.

LLVM has many available frontends for many programming languages too, like C, Rust, Go, C++, and Ada, that emit IR for the backend after language-dependent optimizations. The basic data types in the IR are integers and floats, and there are five built-in derived types: pointers, arrays, vectors, structures, and functions. If the language targeted by the frontend supports more data types, they are expressed in the IR as a combination of the standard IR datatypes.

The first-class citizen frontend of LLVM is CLANG, the C, C++, Objective-C, and Objective-C++ frontend.

We built our prototype for C/C++ programs, so we rely on CLANG as frontend.

The structure of the intermediate representation is SSA, as written before, that makes use of an infinite set of registers. The IR is also strongly typed and RISC-like.

Example.
An example of human-readable IR function is the following:

```
define i32 @max(i32 %x, i32 %y) {
entry:
  %cmp = icmp sgt i32 %x, %y
  br i1 %cmp, label %if.then, label %if.else

if.then:
  br label %if.end

if.else:
  br label %if.end

if.end:
  %m.0 = phi i32 [ %x, %if.then ], [ %y, %if.else ]
  ret i32 %m.0
}
```

It is the translation of the SSA pseudocode discussed in Sec. 4.1.
All the variables are assigned only once and the main difference with the SSA pseudocode is the missing definition of the intermediate `m` variables. LLVM does not support definitions of variables directly using another one because you can just use the original variable (`%x` and `%y` in our case) instead.

The LLVM infrastructure is modular: you can write a so-called LLVM pass that can manipulate the IR and it is invoked during various stages of the IR optimization. Passes can be in-tree, modifying the source tree of LLVM itself like the AddressSanitizer pass, or out-of-tree, that are shared object loaded at runtime.

LLVM incorporates also a debugger, a C++ standard library implementation, and a linker to enable link-time optimizations.

5.2 The Daikon invariant detector

Daikon is a dynamic miner of likely invariants, previously introduced in 2.5.4.

It relies on execution traces that are in a language-independent form, enabling the tool to operate on data produced by different tracers, most notably the Java tracer and the C tracer based on Valgrind [71], and even fictional traces that are not outcomes of the execution of a program.

It requires a declaration (`decls`) file that describes the traced variables and one or more data trace (`dtrace`) files that report the actual values of the traced variables.

In both formats variables are grouped by the program point in which they are observed, typically function enter and exit.

The pattern for a program point entry in the declaration file is[1]:

```
ppt <ppt-name>
<ppt-info>
<ppt-info>
...
variable <name-1>
  <variable-info>
  <variable-info>
  ...
variable <name-2>
  ...
```

The `ppt-name` encodes the name plus its type, for instance `name:::ENTER` for variables when entering a function, `name:::EXIT` for variables when exiting a function, or `name:::OBJECT` for object fields. `ppt-info` encodes properties such as if a method is private and the parent program point in the `dataflow hierarchy` (e.g. an object program point is the parent of all the method program points of such object because a method can access the fields).

The variable names must be unique inside the program point, and the `variable-info` holds information about the data type of the variable, the comparability (Sec. 4.6.1), the bounds if any, if it is a function parameter and other generic flags.

In a data trace, the corresponding program point entry has the following format:

[1]`https://plse.cs.washington.edu/daikon/download/doc/developer/File-formats.html#Program-point-declarations`

```
<program-point-name>
this_invocation_nonce
<nonce-string>
<var-name-1>
<var-value-1>
<var-modified-1>
<var-name2>
<var-value-2>
<var-modified-2>
...
```

The nonce is a progressive number to define a total order between such entries. The logged values for each variable are followed by the `var-modified` field, which tells if the variable was modified since the last time it was logged.

5.3 The AFL++ fuzzing framework

American Fuzzy Lop ++ (AFL++) [37] is a recent fork of the popular coverage-guided fuzzer AFL, that is not improved anymore with novel features from 2017.

AFL++ incorporates and sometimes reimplements some of the latest relevant research in Fuzz Testing, such as [64], [11] and [20].

It supports many different instrumentation backends to extract coverage information from the target, for both compiler-based instrumentation and binary-only instrumentation.

To instrument a program during the compilation pipeline, AFL++ ships a GCC plugin and a set of LLVM passes. For binary-only targets, AFL++ has two Dynamic Binary Translation backends that instrument the code during the recompilation stage of the JIT, one based on QEMU [14] and the other based on Unicorn Engine [72].

It supports different types of code coverage, such as standard edge coverage, context-sensitive edge coverage, ngram coverage, and more. The standard edge coverage is logged in a shared map (`__afl_area_ptr`) used as an observation channel. The number of hits for each edge is logged too.

The classic instrumentation, inherited from AFL, inserts a snippet similar to the following C code at each basic block:

```
void afl_maybe_log(unsigned cur_loc) {

  static __thread unsigned prev_loc = 0;
  __afl_area_ptr[cur_loc ^ prev_loc]++;
  prev_loc = cur_loc >> 1;

}
```

When using the compiler wrapper that uses the LLVM passes to instrument the code, the equivalent of this code is inserted inline at the IR level.

The `cur_loc` parameter is generated at compilation time and identifies the basic block while `prev_loc` maintains the information about the last executed block. So the shared map index `cur_loc ^ prev_loc` is related to the currently executed edge.

5.4 The InvsCov pipeline

We implemented our technique in a prototype called InvsCov that stands for Invariants Coverage.

The InvsCov pipeline is composed of several stages:

1. Dumper compilation: a first version of the program that dumps variables values is compiled;

2. Online learning: the dumper program is executed for each input in the corpus and Daikon performs online learning of the likely invariants;

3. Checks generation: the output of Daikon is processed and an object file with all the checks for each invariant is produced;

4. Target compilation: a second version of the program is compiled instrumenting the blocks with the classic AFL instrumentation plus the calls to the checks (generated C functions) needed for the invariants coverage;

5.4.1 Dumper compilation

The dumper instrumentation is handled by an LLVM pass on functions and a runtime object. This pass is the one that implements Algorithm 5 (Sec. 4.6.1) and also uses Range Analysis to learn the bounds of the integer IR values, when possible.

As an implementation choice, in this pass we reduced the number of IR values considered as variables for the invariants miner if at least one of the following properties hold:

- the value can be directly connected to a local variable in the source code using debug symbols;
- the value is a not a constant index of the pointer operand of a GetElementPtr instruction [2];
- the value is related to a Load or Store instruction (both pointers and values);
- the value is the return value of the function;

During the compilation, the pass dumps the information about program points and variables, such as type, comparability, and bounds, in a JSON file for each module. Then, each file is processed and merged to produce the declaration file for Daikon.

When executed, the dumper binary outputs a dtrace file related to the execution.

Example.
Consider this example function in C language:

```
int isgreater(int x, int y) {

  if (x > y)
    return 1;
  return 0;

}
```

Converted to LLVM IR using Clang it is:

```
define dso_local i32 @isgreater(i32 %x, i32 %y)
    local_unnamed_addr #0 !dbg !16 {
entry:
  call void @llvm.dbg.value(metadata i32 %x, metadata !20,
                            metadata !DIExpression()), !dbg !22
  call void @llvm.dbg.value(metadata i32 %y, metadata !21,
                            metadata !DIExpression()), !dbg !22
  %cmp = icmp sgt i32 %x, %y, !dbg !23
  %. = zext i1 %cmp to i32, !dbg !22
  ret i32 %., !dbg !25
}
```

[2] https://llvm.org/doxygen/classllvm_1_1GetElementPtrInst.html

The `llvm.dbg.value` intrinsic calls for the values %x and %y tell that these values are related to some variables in the source code, x and y.
In this small example, the instrumented values are %x and %y, because they are related to the source code variables, and %., because it is the return value.
The program points in which each variable is logged are basic blocks and not functions. This is trivial to achieve just generating the decl and the dtrace files like if each basic block is a separate function, logging two times the variables at the end of the block to comply with the Daikon formats that need an `ENTER` and at least one `EXIT`.

5.4.2 Online learning

We patched Daikon version 5.8.3 to enable online learning with the dumper binary. Logging all the dtrace files produced by the dumper binary for each input can be expensive in terms of disk space. Instead, we run the dumper inside Daikon and process the outcome on the fly.

After the learning, the invariants are saved as textual output in order to be processed in the next stage.

5.4.3 Checks generation

The file with the invariants that are generated by Daikon is then parsed in this stage.

This file is textual, and for each program point lists the relations, that are expressions over one or more variables.

A simple one can be just `LOC_x > 1` but more complex invariants are possible, such as `LOC_x^2 + 3 * LOC_y - LOC_z >= 0`.

We parse this textual representation using regular expressions to locale the variable names and substitute them with C variables when generating C functions for each check.

The generated function is named `__daikon_constr_ID` where ID is a unique number identifying the invariant.

The return value is the ID shifted by 1 if the invariant is violated, 0 otherwise.

Example.
A generated function looks like the following C snippet:

```
uint32_t __daikon_constr_123(int32_t v0) {

  if (!(v0 > 1))
    return 123 << 1;
  return 0;

}
```

In this case, the ID is 123.

The script that generates the code creates also a JSON file describing each generated function for the next stage.

5.4.4 Target compilation

The final binary, ready to be fuzzed by AFL++, is compiled using an LLVM pass that takes into account the outcome of the checks generation phase.

The LLVM values involved in the invariants are retrieved using their name that is the same between the dumper pass and the present one.

In this pass, the `prev_loc` variable of the AFL++ instrumentation that tracks the incoming block when logging an edge is XOR-ed with the return value of each check function. When the invariant is not violated, the function returns 0, which results in normal edge coverage.

At the end of each block, the inserted instrumentation looks like the one in the following pseudocode:

```
__afl_area_ptr[cur_loc ^ prev_loc]++;
prev_loc = cur_loc >> 1;
prev_loc ^= __daikon_constr_123(variable1);
prev_loc ^= __daikon_constr_321(variable2, variable3);
...
```

Whenever an invariant is shared between two blocks and one dominates the other, the dominated block does not call again the function that performs the check, but directly reuses the outcome of the previously called function in the dominator.

All the instrumentation code inserted by the pass is marked with the `nosanitize` metadata to avoid to be instrumented by sanitizers.

Example.
At the end of the pipeline, the `isgreater` function used in the previous examples, is compiled to LLVM IR as follows:

```
define dso_local i32 @isgreater(i32 %x, i32 %y)
    local_unnamed_addr #0 !dbg !7 {
entry:
  ; AFL++ instrumentation
  %0 = load i32, i32* @__afl_prev_loc, !dbg !14, !nosanitize !2
  %1 = load i8*, i8** @__afl_area_ptr, !dbg !14, !nosanitize !2
  %2 = xor i32 %0, 2620, !dbg !14
  %3 = getelementptr i8, i8* %1, i32 %2, !dbg !14
  %4 = load i8, i8* %3, !dbg !14, !nosanitize !2
  %5 = add i8 %4, 1, !dbg !14
  %6 = icmp eq i8 %5, 0, !dbg !14
  %7 = zext i1 %6 to i8, !dbg !14
  %8 = add i8 %5, %7, !dbg !14
  store i8 %8, i8* %3, !dbg !14, !nosanitize !2
  store i32 1310, i32* @__afl_prev_loc, !dbg !14
  ; actual isgreater code
  call void @llvm.dbg.value(metadata i32 %x, metadata !12,
      metadata !DIExpression()), !dbg !14
  call void @llvm.dbg.value(metadata i32 %y, metadata !13,
      metadata !DIExpression()), !dbg !14
  %cmp = icmp sgt i32 %x, %y, !dbg !15
  %. = zext i1 %cmp to i32, !dbg !14
  ; check invariant and update prev_loc
  %9 = call i32 @__daikon_constr_1(i32 %.), !dbg !17
  %10 = load i32, i32* @__afl_prev_loc, !dbg !17, !nosanitize !2
  %11 = xor i32 %10, %9, !dbg !17
  store i32 %11, i32* @__afl_prev_loc, !dbg !17, !nosanitize !2
  ret i32 %., !dbg !17
}
```

There is a single invariant, checked by `__daikon_constr_1`, and the return value of this function is XOR-ed with `prev_loc`.

Chapter 6

Evaluation

In this chapter, we present the results of a preliminary experimental investigation of our prototype InvsCov. We choose as metrics the heuristically *triaged bugs* in a given time window to evaluate the ability to catch faults, and *executions per second* to evaluate the overhead.

6.1 Setup and Dataset

All the experiments in this chapter were run on an x86_64 machine with the Intel(R) Xeon(R) Platinum 8160 CPU at 2.10GHz and 32 GiB of RAM. The operating system used was Ubuntu 18.04 with the kernel version 4.15.

We selected a set of real-world applications to evaluate our prototype InvsCov. Table 6.1 lists the chosen programs. In the first part there are real-world programs, often old versions, that may contain bugs with a high probability. Most of them were part of evaluations in previous works from the literature (e.g. [84] [67]). In the second part, there are targets with known vulnerabilities taken from Fuzzer Test Suite [44], which are old versions of real-world programs too.

We report in Table 6.1 the list of the programs, their versions, and the sanitizers used during the compilation of the subjects. The missing usage of UBSan on some programs is due to shallow bugs in those subjects that make them crash even with simple valid inputs. In this case, we opted to remove the sanitizer to allow the usage of the program in our evaluation.

In Table 6.2 we list the command line parameters used to run the programs. The subjects from Fuzzer Test Suite are not in this table because all of them use a LibFuzzer harness from OSS-Fuzz.

As described in Sec. 4.7, the choice of the initial corpus for InvsCov not trivial.

Program	Version	Sanitizers
jasper	2.0.16	ASan, UBSan
autotrace	4333e37d5040881b19c2e1dad221f8e988419932	ASan, UBSan
cflow	8a75c3721fd38f8d278cd71fb3682ead1497bb46	ASan, UBSan
catppt (catdoc)	0.95	ASan, UBSan
xls2csv (catdoc)	0.95	ASan, UBSan
pdf2cairo (poppler)	53368f1717e88e40fe65d27e919c9abca11beac3	ASan, UBSan
potrace	1.16	ASan, UBSan
pspp	53d339111a9f51561cfccc65764874cdf54e501a	ASan
exiv2	356f8627371e10cb8719eba3c45789e67420b10a	ASan, UBSan
sndfile (libsndfile)	2ccb23fe724d1d946b4e0c51b791cc655da6962e	ASan, UBSan
lcms	f9d75ccef0b54c9f4167d95088d4727985133c52	ASan, UBSan
re2	499ef7eff7455ce9c9fae86111d4a77b6ac335de	ASan, UBSan
boringssl	894a47df2423f0d2b6be57e6d90f2bea88213382	ASan, UBSan
guetzli	9afd0bbb7db0bd3a50226845f0f6c36f14933b6b	ASan, UBSan
libxml2	v2.9.2	ASan, UBSan
woff2	9476664fd6931ea6ec532c94b816d8fbbe3aed90	ASan
libarchive	51d7afd3644fdad725dd8faa7606b864fd125f88	ASan
pcre2	183	ASan, UBSan

Table 6.1. Target programs versions and sanitizers.

Program	Command line
jasper	-f @@ -t jp2 -T mif -F /dev/null
cflow	–no-main @@
catppt	@@
xls2csv	@@
pdf2cairo	-tiff @@ out
potrace	-b pdf -c -q @@ -o /dev/null
pspp	-O format=txt -o /dev/null -b @@
exiv2	- (OSS-Fuzz harness)
sndfile	–cart –instrument –broadcast @@

Table 6.2. Command line used to run the target programs.

We opted to simply set 12 hours as the time window and run AFL to produce the corpus. This choice, of course, is not optimal in general but allows us to evaluate the technique on a large set of programs, reducing the manual work needed to observe when a fuzzer saturates for every single target. We leave the evaluation with incremental fuzzing after saturation with InvsCov to future work.

All the benchmarks in this chapter are run with a time window of 48 hours and each experiment was repeated 3 times. The reported numbers are the median values.

6.2 Efficiency in finding faults

Program	AFL++ bugs	AFL++InvsCov bugs	Intersection
jasper	27	28	20
autotrace	30	28	28
cflow	6	6	4
catppt	4	8	4
xls2csv	19	29	18
pdf2cairo	25	27	21
potrace	1	0	0
pspp	34	17	12
exiv2	45	50	34
sndfile	18	21	18
lcms	0	1	0
re2	1	0	0
boringssl	6	6	6
guetzli	3	1	1
libxml2	16	17	13
woff2	2	3	2
libarchive	0	0	0
pcre2	104	122	57
Total	341	364	238

Table 6.3. Triaged bugs found during the 48h experiments.

In this section, we evaluate the ability of InvsCov to find more or different bugs than the baseline AFL++ in a given time window of 48 hours.

As the number of reported crashes by the two fuzzers is high, we opted to automatically triage the crashes to the number of bugs heuristically using the hash of the call stack [1] registered when the program crashes. While this is a less sound metric than counting ground-truth bugs (e.g. [48]), it was used in recent past works like [67] successfully.

To be more sound, we removed the addresses belonging to the C and the C++ standard libraries from the call stacks to reduce false positives.

In Table 6.3 we report the number of uncovered bugs for AFL++ and AFL++ with InvsCov. We report the intersection between the sets too, as a fuzzer that finds less but different bugs than another are still interesting.

6.3 Performance overhead

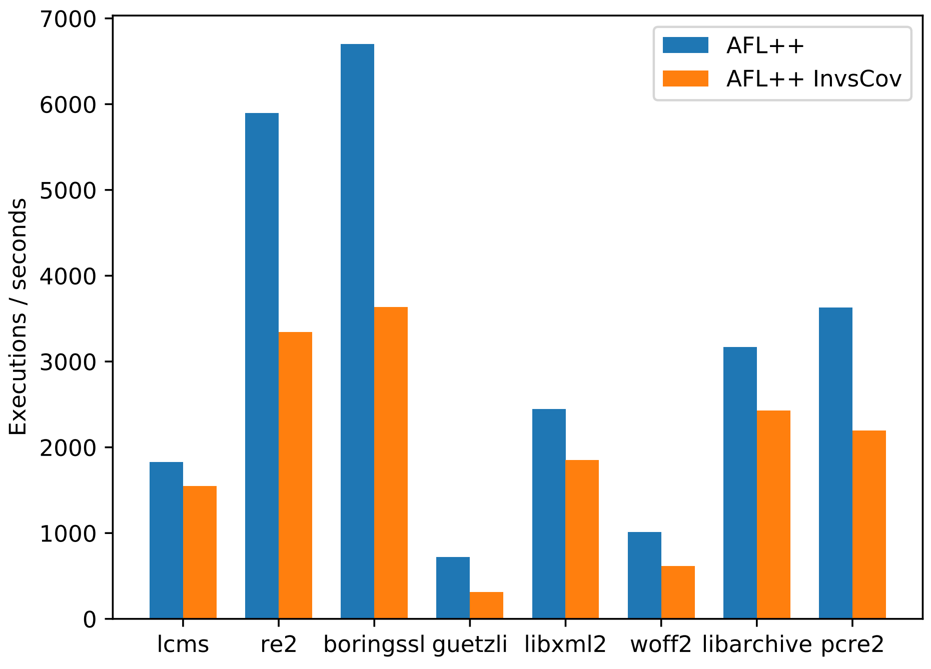

Figure 6.1. Comparison in medium speed over a 48h experiment.

In this section, we evaluate the performance overhead in terms of speed.

We compare the medium speed registered by the fuzzer (executions per second) for an entire 48-hour run. The chosen subjects are the targets used in Sec. 6.2 that come from Fuzzer Test Suite. These targets use the harnesses from OSS-Fuzz and so

[1]The hash represents the sequence of functions that are concurrently active on the run-time stack at a given moment [28].

AFL++ can execute them in Persistent Mode, allowing a more sound comparison in terms of speed without the overhead of `fork()` operations.

In Figure 6.1 the medium executions per second of AFL++ and AFL++InvsCov are compared. The gap is almost never more than 2x, even smaller for targets with medium speed (2000 execs/sec).

The medium overhead over the set of benchmarks is 1.62x.

6.4 Discussion

From the evaluation of our technique for the ability to trigger faults, it appears that our technique performs well when the coverage produced by the initial corpus already covers most of the program. In targets like potrace, our technique fails to uncover a bug that is in a code region not covered by the initial corpus. On other targets, such as pcre2, InvsCov performs well and uncovers more and different faults than vanilla AFL++.

This highlights that the limitation in speed showed in the second part of the evaluation, can sometimes decrease the performance in uncovering bugs in new program points. This can be considered a limitation, or not because the purpose of the technique is not to be a replacement for Coverage-guided Fuzzing, but an incremental step instead. The problem can be addressed with a wiser choice of the initial corpus, taking the testcases of an already saturated fuzzer such as AFL++ itself.

The overall results suggest that InvsCov improves the state-of-the-art of Fuzz Testing in terms of findings faults especially when the code is already explored by the initial corpus.

Chapter 7

Conclusion

In this thesis, we introduced a new feedback for Feedback-driven Fuzz Testing in order to approximate the program state coverage better than traditional Coverage-guided Fuzzing.

Reaching a program point does not guarantee the discovery of a fault in such a portion of code: we proposed to distinguish the same program point in the registered coverage if the values in the program state are unusual using likely invariants.

The likely invariants are mined local constraints that, if violated, may uncover the presence of a bug or a local unusual state. In both cases, they define a meaningful division of the possible values used in the corresponding program point.

In our technique, we learn invariants at the granularity of basic blocks to define a feedback that combines edge coverage and invariants violations.

In this way, we were able to augment the sensitivity of the coverage feedback of Feedback-driven Guided Fuzzing taking into account data and not only code without the typical path explosion issue that characterizes coverage types based on data tracking.

The developed prototype, InvsCov, extends the compilation pipeline of LLVM to produce binaries that can be used to learn invariants and to record invariants and code coverage for AFL++.

We showed that the prototype works on a set of real-world benchmarks producing fully functional binaries that can be easily fuzzed uncovering more of different types of faults than vanilla AFL++ with a reasonable performance overhead.

Based on our results, in general, we can devise that augmenting the sensitivity of a feedback, automatically and not just manually for a small set of program points and variables, with a sane amount of useful information can improve the search algorithm of the fuzzer.

7.1 Future directions

We foresee two directions of improvement: one is technical, the other concerns the implementation.

Firstly, our methodology can grow to learn the invariants on-demand during the run of the fuzzer. Instead of being a preprocessing step before the run of the fuzzer, we can adapt it to modify the target program on-the-fly with discovered likely invariants. In this way, we can use our technique on newly discovered code too, and not only on program points that are already covered by the execution of the input corpus.

Another avenue for technical improvement is the tracking of the side effect in the memory state. We employ such abstraction to avoid logging each value in the memory state, which would be an impossible solution in practice. In the future, we can augment such a method to not only track the values in the basic block state, which contains the side effects in memory, but also other interesting values that are implicitly related to the program state but not directly used in the block. For instance, consider the size of a buffer in the heap that is not used as a variable in a basic block that simply performs access to such buffer. The buffer size is still an interesting value to track because it has an indirect relationship with the memory address used in the block.

On the implementation side, we should replace the Daikon invariant detector with a more performant engine that uses the GPU. Daikon is a mono thread program in Java that runs on the CPU, while its algorithm can be reimplemented to exploit the parallelization opportunities from modern machines.

Another future direction is the implementation of InvsCov on the IR of an emulator. The LLVM implementation rarely uses information about the source code and so our technique, that does not depend on the particular frontend language, can be easily reimplemented as an instrumentation pass of a JIT compiler of a dynamic binary translator like QEMU.

Bibliography

[1] Basic Blocks (GNU Compiler Collection (GCC) Internals). [Online; accessed 10-July-2020]. Available from: `https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html`.

[2] IEEE Guide for Software Verification and Validation Plans. *IEEE Std 1059-1993*, (1994), 1.

[3] Circumventing Fuzzing Roadblocks with Compiler Transformations. `https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/` (2016). [Online; accessed 10-Sep-2020].

[4] ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. *C/S2ESC - Software Systems Engineering Standards Committee*, (2017).

[5] Google OSS-Fuzz: continuous fuzzing of open source software. `https://github.com/google/oss-fuzz` (2019).

[6] Allen, F. E. Control flow analysis. *SIGPLAN Not.*, **5** (1970), 1. Available from: `https://doi.org/10.1145/390013.808479`, `doi:10.1145/390013.808479`.

[7] Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A., and Teuchert, D. Nautilus: Fishing for deep bugs with grammars. In *NDSS* (2019).

[8] Aschermann, C. and Schumilo, S. What the fuzz (2019). [Online; accessed 10. Sep. 2020]. Available from: `https://hexgolems.com/talks/blackhat_19.pdf`.

[9] Aschermann, C. and Schumilo, S. On Measuring and Visualizing Fuzzer Performance (2020). [Online; accessed 1. Sep. 2020]. Available

from: `https://hexgolems.com/2020/08/on-measuring-and-visualizing-fuzzer-performance/`.

[10] Aschermann, C., Schumilo, S., Abbasi, A., and Holz, T. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)* (2020).

[11] Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., and Holz, T. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS* (2019). Available from: `https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/`.

[12] Aumasson, J.-P. and Romailler, Y. Automated testing of crypto software using differential fuzzing. *Black Hat USA, Jul*, (2017).

[13] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys*, **51** (2018), 50:1. Available from: `http://doi.acm.org/10.1145/3182657`, `doi:10.1145/3182657`.

[14] Bellard, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pp. 41–41. USENIX Association, Berkeley, CA, USA (2005). Available from: `http://dl.acm.org/citation.cfm?id=1247360.1247401`.

[15] Blanchet, B. Introduction to abstract interpretation. (2002).

[16] Blazytko, T., Aschermann, C., Schlögel, M., Abbasi, A., Schumilo, S., Wörner, S., and Holz, T. GRIMOIRE: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1985–2002. USENIX Association, Santa Clara, CA (2019). ISBN 978-1-939133-06-9. Available from: `https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko`.

[17] Böhme, M. and Falk, B. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, pp. 1–12 (2020).

[18] Böhme, M., Manès, V., and Cha, S. K. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 14th Joint meeting*

of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE, pp. 1–11 (2020).

[19] Böhme, M. and Paul, S. A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, **42** (2016), 345.

[20] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pp. 1032–1043. Association for Computing Machinery, New York, NY, USA (2016). ISBN 9781450341394. Available from: `https://doi.org/10.1145/2976749.2978428`, `doi:10.1145/2976749.2978428`.

[21] Claessen, K. and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pp. 268–279. Association for Computing Machinery, New York, NY, USA (2000). ISBN 1581132026. Available from: `https://doi.org/10.1145/351240.351266`, `doi:10.1145/351240.351266`.

[22] Clements, A. A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., and Payer, M. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1201–1218. USENIX Association (2020). ISBN 978-1-939133-17-5. Available from: `https://www.usenix.org/conference/usenixsecurity20/presentation/clements`.

[23] Cova, M., Balzarotti, D., Felmetsger, V., and Vigna, G. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pp. 63–86. Queensland, Australia (2007).

[24] Csallner, C. and Smaragdakis, Y. Check'n'crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering*, pp. 422–431 (2005).

[25] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control depen-

dence graph. *ACM Trans. Program. Lang. Syst.*, **13** (1991), 451. Available from: `https://doi.org/10.1145/115372.115320`, `doi:10.1145/115372.115320`.

[26] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, **13** (1991), 451. Available from: `https://doi.org/10.1145/115372.115320`, `doi:10.1145/115372.115320`.

[27] D'Elia, D. C. and Demetrescu, C. On-stack replacement, distilled. *SIGPLAN Not.*, **53** (2018), 166. Available from: `https://doi.org/10.1145/3296979.3192396`, `doi:10.1145/3296979.3192396`.

[28] D'Elia, D. C., Demetrescu, C., and Finocchi, I. Mining hot calling contexts in small space. *Software: Practice and Experience*, **46** (2016), 1131. Available from: `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2348`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2348`, `doi:10.1002/spe.2348`.

[29] Dinesh, S., Burow, N., Xu, D., and Payer, M. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *IEEE S&P 2020* (2020).

[30] Eddington, M. Peach fuzzing platform. `https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html`. [Online; accessed 10-Sep-2020].

[31] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, **27** (2001), 99.

[32] Ernst, M. D. and Notkin, D. *Dynamically Discovering Likely Program Invariants.* Ph.D. thesis, USA (2000). AAI9983472.

[33] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, **69** (2007), 35.

[34] Falk, B. Brandon Falk @gamozolabs FuzzBench feedback (2020). [Online; accessed 1. Sep. 2020]. Available from: `https://github.com/google/fuzzbench/issues/654`.

[35] Fioraldi, A., D'Elia, D. C., and Coppa, E. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020. Association for Computing Machinery, New York, NY, USA (2020). ISBN 9781450380089. Available from: `https://doi.org/10.1145/3395363.3397372`, `doi:10.1145/3395363.3397372`.

[36] Fioraldi, A., D'Elia, D. C., and Querzoni, L. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)* (2020).

[37] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association (2020).

[38] Floyd, R. W. Assigning meanings to programs. In *Program Verification*, pp. 65–81. Springer (1993).

[39] Frighetto, A. Fuzzing binaries with llvm's libfuzzer and rev.ng (2020). [Online; accessed 20. Sep. 2020]. Available from: `https://rev.ng/blog/page-1.html`.

[40] Giuffrida, C., Cavallaro, L., and Tanenbaum, A. Practical automated vulnerability monitoring using program state invariants. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 1–12. IEEE CS (2013). ISBN 978-1-4673-6471-3. `doi:10.1109/DSN.2013.6575318`.

[41] Godefroid, P. 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)*, pp. 1–1 (2007).

[42] Godefroid, P., Levin, M. Y., and Molnar, D. Automated whitebox fuzz testing (2008). Available from: `https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/`.

[43] Godefroid, P., Peleg, H., and Singh, R. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pp. 50–59. IEEE

Press, Piscataway, NJ, USA (2017). ISBN 978-1-5386-2684-9. Available from: http://dl.acm.org/citation.cfm?id=3155562.3155573.

[44] Google. Fuzzer test suite. [Online; accessed 1. Sep. 2020]. Available from: https://github.com/google/fuzzer-test-suite.

[45] Groce, A. and Regehr, J. The Saturation Effect in Fuzzing. https://blog.regehr.org/archives/1796.

[46] Hangal, S. and Lam, M. S. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pp. 291–301. Association for Computing Machinery, New York, NY, USA (2002). ISBN 158113472X. Available from: https://doi.org/10.1145/581339.581377, doi:10.1145/581339.581377.

[47] Harrison, W. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, **3** (1977), 243. doi:10.1109/TSE.1977.231133.

[48] Hazimeh, A., Herrera, A., and Payer, M. Magma: A ground-truth fuzzing benchmark. (2020). arXiv:2009.01120.

[49] Hoare, C. A. Proof of correctness of data representations. *Acta Inf.*, **1** (1972), 271. Available from: https://doi.org/10.1007/BF00289507, doi:10.1007/BF00289507.

[50] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Comm. ACM*, **12** (1969), 576.

[51] Holler, C., Herzig, K., and Zeller, A. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pp. 445–458. USENIX Association, Bellevue, WA (2012). ISBN 978-931971-95-9. Available from: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[52] Huang, H., Yao, P., Wu, R., Shi, Q., and Zhang, C. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1613–1627. IEEE Computer Society, Los Alamitos, CA, USA (2020). Available from: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00063, doi:10.1109/SP40000.2020.00063.

[53] Jonathan Metzman, L. S., Abhishek Arya. FuzzBench: Fuzzer benchmarking as a service. Google Security Blog (2020).

[54] Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pp. 2123–2138. Association for Computing Machinery, New York, NY, USA (2018). ISBN 9781450356930. Available from: `https://doi.org/10.1145/3243734.3243804`, `doi:10.1145/3243734.3243804`.

[55] Lamport, L. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (1977), 125.

[56] Lattner, C. *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (2002). *See* `http://llvm.cs.uiuc.edu`.

[57] Lattner, C. and Adve, V. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*. West Lafayette, Indiana (2004).

[58] Laycock, G. T. *The theory and practice of specification based software testing*. Ph.D. thesis, Citeseer.

[59] Le Lann, G. An analysis of the ariane 5 flight 501 failure - a system engineering perspective. In *Proceedings of the 1997 International Conference on Engineering of Computer-Based Systems*, ECBS'97, pp. 339–346. IEEE Computer Society, USA (1997). ISBN 0818678895.

[60] Lemieux, C., Padhye, R., Sen, K., and Song, D. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pp. 254–265. Association for Computing Machinery, New York, NY, USA (2018). ISBN 9781450356992. Available from: `https://doi.org/10.1145/3213846.3213874`, `doi:10.1145/3213846.3213874`.

[61] Leveson, N. G. and Turner, C. S. An investigation of the therac-25 accidents. *Computer*, **26** (1993), 18. Available from: `https://doi.org/10.1109/MC.1993.274940`, `doi:10.1109/MC.1993.274940`.

[62] LLVM Project. LibFuzzer - Value Profile. Available from: `https://llvm.org/docs/LibFuzzer.html#value-profile`.

[63] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. (2018). Available from: https://llvm.org/docs/LibFuzzer.html.

[64] Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y., and Beyah, R. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1949–1966. USENIX Association, Santa Clara, CA (2019). ISBN 978-1-939133-06-9. Available from: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

[65] Maier, D., Radtke, B., and Harren, B. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, Santa Clara, CA (2019). Available from: https://www.usenix.org/conference/woot19/presentation/maier.

[66] Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, **xxx** (2019), xxx.

[67] Manès, V. J. M., Kim, S., and Cha, S. K. Ankou: Guiding grey-box fuzzing towards combinatorial difference. pp. 1024–1036 (2020).

[68] Mathis, B., Gopinath, R., and Zeller, A. Learning input tokens for effective fuzzing. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020* (edited by S. Khurshid and C. S. Pasareanu), pp. 27–37. ACM (2020). Available from: https://doi.org/10.1145/3395363.3397348, doi:10.1145/3395363.3397348.

[69] Maurício Aniche. Software Testing: From Theory to Practice (2020). Available from: https://sttp.site/.

[70] Muench, M., Stijohann, J., Kargl, F., Francillon, A., and Balzarotti, D. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*. San Diego, UNITED STATES (2018). Available from: http://www.eurecom.fr/publication/5417.

[71] Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007*

Conference on Programming Language Design and Implementation (PLDI 2007), pp. 89–100. San Diego, California, USA (2007).

[72] Ngyuen, A. Q. and Dang, H. V. Unicorn: Next generation cpu emulator framework (2020). Available from: `http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf`.

[73] Österlund, S., Razavi, K., Bos, H., and Giuffrida, C. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security* (2020). Available from: `Paper=https://download.vusec.net/papers/parmesan_sec20.pdfCode=https://github.com/vusec/parmesan`.

[74] Padhye, R., Lemieux, C., Sen, K., Papadakis, M., and Le Traon, Y. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pp. 329–340. Association for Computing Machinery, New York, NY, USA (2019). ISBN 9781450362245. Available from: `https://doi.org/10.1145/3293882.3330576`, `doi:10.1145/3293882.3330576`.

[75] Padhye, R., Lemieux, C., Sen, K., Simon, L., and Vijayakumar, H. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, **3** (2019). Available from: `https://doi.org/10.1145/3360600`, `doi:10.1145/3360600`.

[76] Pattabiraman, K., Saggese, G. P., Chen, D., Kalbarczyk, Z., and Iyer, R. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, **8** (2010), 640.

[77] Peng, H. and Payer, M. Usbfuzz: A framework for fuzzing USB drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2559–2575. USENIX Association (2020). ISBN 978-1-939133-17-5. Available from: `https://www.usenix.org/conference/usenixsecurity20/presentation/peng`.

[78] Peng, H., Shoshitaishvili, Y., and Payer, M. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 697–710 (2018). Available from: `https://doi.org/10.1109/SP.2018.00056`, `doi:10.1109/SP.2018.00056`.

[79] Pham, V., Boehme, M., Santosa, A. E., Caciulescu, A. R., and Roychoudhury, A. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, (2019). doi:10.1109/TSE.2019.2941681.

[80] Pham, V., Böhme, M., and Roychoudhury, A. Aflnet: A greybox fuzzer for network protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track* (2020).

[81] Poeplau, S. and Francillon, A. Symbolic execution with symcc: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 181–198. USENIX Association (2020). ISBN 978-1-939133-17-5. Available from: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

[82] Prosser, R. T. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pp. 133–138. Association for Computing Machinery, New York, NY, USA (1959). ISBN 9781450378680. Available from: https://doi.org/10.1145/1460299.1460314, doi:10.1145/1460299.1460314.

[83] Quintao Pereira, F. M., Rodrigues, R. E., and Sperle Campos, V. H. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pp. 1–11. IEEE Computer Society, USA (2013). ISBN 9781467355247. Available from: https://doi.org/10.1109/CGO.2013.6494996, doi:10.1109/CGO.2013.6494996.

[84] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS* (2017). Available from: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

[85] Reed Hastings, B. J. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference.* Citeseer (1991).

[86] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, **74** (1953), 358. Available from: http://www.jstor.org/stable/1990888.

[87] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pp. 12–27. Association for Computing Machinery, New York, NY, USA (1988). ISBN 0897912527. Available from: `https://doi.org/10.1145/73560.73562`, `doi:10.1145/73560.73562`.

[88] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. Kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pp. 167–182. USENIX Association, USA (2017). ISBN 9781931971409.

[89] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, p. 28. USENIX Association (2012).

[90] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, vol. 16, pp. 1–16 (2016).

[91] Swiecki, R. Honggfuzz. [Online; accessed 1. Sep. 2020]. Available from: `https://github.com/google/honggfuzz`.

[92] Tillmann, N., Chen, F., and Schulte, W. Discovering likely method specifications. In *International Conference on Formal Engineering Methods*, pp. 717–736. Springer (2006).

[93] Vranken, G. VrankenFuzz a multi-sensor, multi-generator mutational fuzz testing engine. `https://guidovranken.files.wordpress.com/2018/07/vrankenfuzz.pdf` (2018).

[94] Vyukov, D. syzkaller - kernel fuzzer. [Online; accessed 10. Sep. 2020]. Available from: `https://github.com/google/syzkaller`.

[95] Wang, J., Duan, Y., Song, W., Yin, H., and Song, C. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pp. 1–15. USENIX Association, Chaoyang District, Beijing

(2019). ISBN 978-1-939133-07-6. Available from: https://www.usenix.org/conference/raid2019/presentation/wang.

[96] Weyuker, E. J. and Jeng, B. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, **17** (1991), 703.

[97] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’11, p. 283âĂŞ294. Association for Computing Machinery, New York, NY, USA (2011). ISBN 9781450306638. Available from: https://doi.org/10.1145/1993498.1993532, doi:10.1145/1993498.1993532.

[98] Yun, I., Lee, S., Xu, M., Jang, Y., and Kim, T. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC’18, pp. 745–761. USENIX Association, USA (2018). ISBN 9781931971461.

[99] Zalewski, M. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt (2016).

[100] Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., and Sun, L. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1099–1114. USENIX Association, Santa Clara, CA (2019). ISBN 978-1-939133-06-9. Available from: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.