LibAFL: A Framework to Build Modular and Reusable Fuzzers

Andrea Fioraldi
EURECOM
fioraldi@eurecom.fr

Dominik Maier
Google
dmnk@google.com

Dongjia Zhang
The University of Tokyo
toka@aflplus.plus

Davide Balzarotti
EURECOM
balzarot@eurecom.fr

ABSTRACT

The release of AFL marked an important milestone in the area of software security testing, revitalizing fuzzing as a major research topic and spurring a large number of research studies that attempted to improve and evaluate the different aspects of the fuzzing pipeline.

Many of these studies implemented their techniques by forking the AFL codebase. While this choice might seem appropriate at first, combining multiple forks into a single fuzzer requires a high engineering overhead, which hinders progress in the area and prevents fair and objective evaluations of different techniques. The highly fragmented landscape of the fuzzing ecosystem also prevents researchers from combining orthogonal techniques and makes it difficult for end users to adopt new prototype solutions.

To tackle this problem, in this paper we propose LibAFL, a framework to build modular and reusable fuzzers. We discuss the different components generally used in fuzzing and map them to an extensible framework. LibAFL allows researchers and engineers to extend the core fuzzer pipeline and share their new components for further evaluations. As part of LibAFL, we integrated techniques from more than 20 previous works and conduct extensive experiments to show the benefit of our framework to combine and evaluate different approaches. We hope this can help to shed light on current advancements in fuzzing and provide a solid base for comparative and extensible research in the future.

1 INTRODUCTION

Fuzzers are tools designed to execute a target application with a large number of automatically-generated inputs. Their goal is to discover problematic states, often associated with the presence of security vulnerabilities. Because of their effectiveness, fuzzers are today an essential asset in the arsenal of both developers and security researchers.

Many off-the-shelf fuzzers are available to the public, some of which are now considered de-facto standards for general-purpose applications: AFL [76], AFL++ [27], HonggFuzz [69], and LibFuzzer [47]. These fuzzers are very popular among security testers and, for example, routinely discover thousands of bugs on OSS-Fuzz [2], an extensive fuzzing effort for open-source software.

Unfortunately, while off-the-shelf fuzzers are great tools that are easy to set up and use for non-experts, they often show their limitations for experienced users. In fact, to test complex applications or to adapt to different types of targets, such as operating systems kernels, device drivers, or embedded devices, experts often resort to creating new fuzzers, or modifying existing ones to fit their needs. For instance, academic researchers often implement algorithmic improvements and new ideas in small prototypes, often built on top of AFL or AFL++. While this satisfies the need of the reviewers in terms of reproducibility of the results, it also resulted in an incredible number of mostly-incompatible forks.

This is due to the fact that all existing fuzzing frameworks are not designed to be extensible. Thus, researchers are forced to reinvent the wheel over and over when implementing their prototypes, often missing out on features that are present in other forks and that are too complex to port or re-implement. Some projects, notably AFL++ [27], proposed highly configurable architectures for fuzzing. However, they are not sufficiently generic (e.g., all inputs are represented as byte arrays, thus requiring hacks and workarounds to integrate structured and grammar fuzzing techniques) nor properly compartmentalized (thus requiring to fork the project to adapt it to new techniques).

This problem is not only an engineering issue, but it also highlights the lack of a standard definition of the entities that define a modern fuzzer. Manes et al. [52] published an academic survey that covers all fuzz testing efforts until 2019. The authors highlight the enormous number of public fuzzers and categorize some common high-level concepts in a generic fuzzing algorithm. While this high-level categorization is sufficient for a systematization, the entities, and their relationships are too coarse-grained to develop a fuzzer framework according to this definition.

The fragmentation of the fuzzing landscape has three critical consequences on the research in the field.

(1) Orthogonal contributions are difficult to combine.
    Several hundred, if not thousands, of different improvements have been proposed in the last decade to increase the effectiveness of fuzz testing. However, a new corpus scheduler implemented on top of AFL cannot be easily combined with a new mutator implemented in a custom fuzzer. As we mentioned before, this hinders the progress of fuzzing as a whole. Each individual tool focuses on a few advanced techniques but cannot take advantage of other orthogonal approaches proposed by other researchers.
(2) Individual contributions are difficult to assess.
    A common drawback of many papers on fuzzing is that the authors compare their technique (for instance, a scheduler) which they implemented on a certain fuzzer, with previously-proposed solutions implemented in different tools. Thus, it is often difficult to understand whether better results are only due to the novel algorithm and not the result of other components of the fuzzer.

(3) Different solutions are difficult to compare.

While dozens of different techniques exist for every aspect of fuzzing, a third-party comparison would require a considerable re-implementation effort, typically reserved for surveys and systematization of knowledge papers. As a result, only a selected amount of solutions have been properly tested and compared on the same datasets.

We believe that these three issues are essential roadblocks that significantly slow down the progress of fuzzing, the transition of new techniques from academia to industry, and the development of new solutions, due to an extensive duplication of work.

Therefore, we propose LIBAFL, a novel fuzzing framework written from scratch in Rust. LIBAFL consists of a collection of libraries that can be used to build custom fuzzers by combining components based on extensible entities. It achieves this goal thanks to several factors: (a) it is based on a novel categorization of components of modern fuzzers; (b) it is designed to exploit the features of Rust, such as easy and fast serialization of objects and component slotting at compile-time; (c) it implements a wide range of fuzzing algorithms, features, and instrumentation options proposed by recent works in the field, and (d) it is easily extensible.

LIBAFL's building blocks can be used to recreate several modern fuzzing solutions. Thanks to the extensible design, researchers can combine blocks and experiment with compositions of beneficial techniques. In this paper, we use a range of building blocks implemented in LIBAFL to combine and test compelling fuzzing approaches that were never before evaluated, and others that were never evaluated on top of the same baseline. To the best of our knowledge, we are the first work to conduct such an extensive re-implementation (we integrated techniques from 20 previous works) and evaluation (15 different techniques) by using the same baseline. Additionally, we evaluate combinations of these techniques and provide insights into the effectiveness of these combinations.

We also show how LIBAFL is a robust base to develop standard and more exotic fuzzers. In the first category, we build a generic bit-level fuzzer that uses an optimal combination of known techniques and show how the result outperforms all state-of-the art generic fuzzers like AFL++, LIBFUZZER, and HONGGFUZZ. We then re-implemented a differential fuzzer for the Ethereum virtual machines [50] by using a custom feedback based on the VM state. We compare this fuzzer with its original implementation, and show how our version outperforms it in terms of uncovered differences in the two tested VMs.

1.1 Contributions

In short, in this paper, we propose the following contributions:

- We identify and model common building blocks used by modern fuzzers;
- We present LIBAFL, a novel open-source fuzzing framework written from scratch in Rust;
- We implement state-of-the-art building blocks and techniques;
- Based on these building blocks, we evaluate 15 techniques proposed in prior work, as well as a range of novel combinations;

- We present a case study that re-implements a differential fuzzer using custom feedbacks;
- Our generic fuzzer outperforms all off-the-shelf fuzzers;

2 BACKGROUND

Fuzz Testing, or fuzzing, has a long history in the area of security testing. Its most naive embodiment, which consisted in looking for crashes by repeatedly feeding random input data to programs, is over 30 years old [55]. From this initial idea, fuzzing has evolved in multiple directions during the past three decades.

These include new techniques to efficiently and effectively explore the target programs, but also the ability to fuzz new systems (such as entire operating systems [41, 51, 65] or hypervisors [64]) or the application of fuzzing beyond its original use in software testing (e.g, for exploit generation [36] or root cause analysis [11, 66]). In their survey, Manes et al. [52] define an algorithm that generalize fuzz testing to accommodate many fuzzers, but their definition was still tied to the notion of bugs, while recent fuzzers evolved to cover applications in other domains. Therefore, we believe it is more appropriate today to think of fuzzing as a *family* of testing techniques, which repeatedly provide machine-generated inputs to a target system with the aim of finding inputs that satisfy certain objectives.

In the same work, the authors consolidate the commonly-adopted taxonomy [29] based on the amount of introspection of the target system required by a fuzzer for each run. The categorization proposes the following three families:

- *Black-box* fuzzers do not need any feedback from the target. In this case, fuzzing is closely related, and maybe even equal, to traditional Random Testing [35]. 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 [22] require a model of the input format to generate testcases;
- *White-box* fuzzers use target-specific, internal information to inspect the state space of the target systematically. An example is SAGE [30] that tries to maximize code coverage by using constraints gathered during the execution;
- *Grey-box* fuzzers stand in the middle of the two previous approaches. They collect minimal information to explore the input space better while keeping the performance overhead low. Traditionally, the information is collected during the target execution with lightweight instrumentation like in AFL [76];

Another common categorization used to describe fuzzer, orthogonal to the previous one, is based on how the fuzzer generates the inputs for each run of the target: (1) *Model-based* generation uses a model of the input format to generate testcases from scratch. This can be a grammar specified by a human (e.g. [39]), a model built by using learning techniques (e.g. [31]), or a set of generation rules hardcoded in the fuzzer algorithm (e.g. [74]); (2) *Mutation-based* generation uses instead previous testcases stored in a *corpus* to derive new inputs. It is often used in gray-box approaches where the corpus evolves thanks to the information collected from the target;

LibAFL: A Framework to Build Modular and Reusable Fuzzers

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

The two approaches are not mutually exclusive. For instance, mutation techniques can use a model of the input format like model-based generation [4, 59]. The difference, in this case, is that the new testcase is not generated from scratch.

A widely popular embodiment of the gray or white box approaches is *evolutionary fuzzing*. It traces the execution of the target by observing some runtime features and then it evolves the internal state by using a feedback based on such observations. For instance, it is common to use the feedback to assess the novelty of an execution and decide whether the new input should be added to the corpus for future mutations.

The most common form of feedback is based on code coverage [17, 27, 47, 65, 69, 76]. Additionally, different types of feedbacks are employed to extend beyond simple coverage-guidance, such as the minimization of the hamming-distance between two values of a comparison instruction [45, 58] or the novelty search in the program state's data [24, 53].

2.1 AFL++

Among the existing solutions, the closest to our work AFL++, the community-driven fork of the American Fuzzy Lop fuzzer and the first project to aggregate different techniques and provide a certain level of extensibility. Fioraldi et al. [27] re-implemented in AFL++ several techniques that the authors considered interesting, such as MOPT [48] and AFLFAST [14], and performed some comparative experiments. Moreover, they defined a plugin interface called *custom mutators* that can be used to extend AFL++ with custom mutations and testcase minimization. It also provides various hooks triggered during the fuzzer lifecycle, for instance, when a testcase gets pulled from the corpus.

While this provided an initial step in the same direction of LibAFL, it suffered from intrinsic limitations derived from the design of AFL. In fact, AFL++ remains a monolithic C codebase for the majority of the tasks that are not related to writing a mutator and different components are not separated by any software engineering criteria. In recent years, in fact, several forks of AFL++ [24, 49, 68, 72] have been developed, continuing the AFL's tradition of incompatible forks implementing orthogonal techniques. Moreover, since the publication of its academic paper, the AFL++ codebase incorporated more and more techniques and increased in size and scope, making it increasing difficulty to maintain the software. To avoid to inherit these problems, we decided to implement LibAFL from scratch starting from a complete new design instead of extending AFL++, even if it would have been a solid base to build upon.

3 ENTITIES IN MODERN FUZZING

To support the design of our framework, we first identified a set of 9 basic entities that are commonly present in most modern fuzzers. In this section, we present these entities and provide some examples by using state-of-the-art fuzzers.

Input – Formally, the input of a program, or a system in general, is the data taken from external sources that affect its behavior. In our model of an abstract fuzzer, we define Input as the internal representation of the program input (or a part of it). In the simplest case, the input of the program is a single-byte array. Fuzzers such as AFL store and manipulate this byte array directly, delivering the result to the target upon execution.

However, there are cases in which a byte array is not an ideal representation of an Input, e.g., when the target expects a sequence of system calls [70]. In this case, a fuzzer does not internally represent the Input in the same way that the program consumes it. Another example is the inputs for grammar fuzzers like NAUTILUS by Aschermann et. al. [4]. Here, the fuzzer internally stores Inputs as Abstract Syntax Trees, a data structure that can be easily manipulated while maintaining validity according to the grammar. Since the target expects a byte array as input, the tree is serialized to a sequence of bytes just before the execution. Other fuzzers may also use other input representations, such as sequences of tokens encoded as integers [63], or the intermediate representation of a programming language [34].

Corpus – The Corpus is a storage for inputs and their associated metadata. Different kind of storage affects the capabilities of a fuzzer, for instance, a corpus that lives entirely in memory makes the fuzzer faster but can quickly exhaust the available memory when fuzzing large targets, while a corpus stored on disk allows the user to inspect the state of the fuzzer but introduce a bottleneck on disk operations.

Most mainstream fuzzers [14, 47, 76] store the corpus on disk, but this choice affects the scalability of parallel fuzzing and requires a standard library to perform the file IO operations.

In our model, a fuzzer requires at least two separate corpora: one that is used to store *interesting* testcases (3) that are used as component of the evolutionary algorithm of the fuzzer, and another one used to store the *solutions*, i.e., the testcases that fulfill the objective of the fuzzer (e.g., program crashes).

Scheduler – The Scheduler is a component tied with the corpus. It is the way the fuzzer asks for the next testcase to fuzz, typically by selecting one entry from the corpus. Naive schedulers implement, for instance, a simple FIFO policy or a random selection. More complex schedulers may use probabilistic algorithms based on introspection statistics about the fuzzer [12] or apply other schedulers to a subset of the corpus, as AFL does when calculating the "favored" minset.

Other examples include schedulers that try to mitigate the explosion of the corpus caused by too sensitive feedback [72] or to prioritize testcases with interesting properties [73].

Stage – The Stage is a component defining an action to perform on a single testcase from the corpus. Usually, the scheduler selects a testcase and then the fuzzer executes every stage on that given input. The Stage is a very broad entity and in existing fuzzers, it is usually the component that invokes one or more times a mutator on the input (e.g., the *random havoc* stage in AFL) or an analysis stage that, for instance, perform taint tracking to gather information in a white-box fuzzer [17].

Another widely known stage adopted by many fuzzers is the minimization phase, introduced in AFL, reduces the size of a testcase obtained from the corpus while maintaining the triggered coverage points.

Observer – The Observer is an entity that provides information from a single execution of the target. To reason on an execution of an input, the fuzzer executes it and then looks at the observers. A snapshot of the observers state after an execution is equivalent to the execution itself in terms of effects on the fuzzer state. Defining the observers in this way is particularly useful when a distributed fuzzer can send the observers state across multiple nodes. This avoids the need for re-execution with the same input when fuzzing a very slow target.

An example of observer is the coverage map used by common coverage-guided fuzzers such as AFL or HonggFuzz that is filled during the execution to report the executed edges. This information is not preserved across runs and it is an observation of a dynamic property of the program.

Other fuzzers, such as Ijon [5] or FuzzFactory [58], use different forms of observers but always rely on a map to keep track of additional metrics beyond code coverage.

Executor – The Executor is the component responsible to execute the target system given an input from the fuzzer. In different fuzzers, the embodiment of this entity may change a lot. For instance, for in-memory fuzzers like LibFuzzer an execution is a call to a harness function, while for hypervisor-based fuzzers like Nyx [64] it requires an entire operating system to re-start from a snapshot at each run.

In our model, the Executor is the entity that defines not only how to execute the target, but all the volatile operations that are related to a single run of the target. So the Executor is, for instance, responsible for informing the program about the input that the fuzzer wants to use in the run, e.g., by writing to a memory location or by passing it as a parameter to the harness function. The Executor also maintains a set of Observers linked with each execution.

Feedback – The Feedback is an entity that classifies the outcome of an execution of the program under test as interesting or not. Typically, this information is used to decide whether the corresponding input is added to a corpus.

Most of the time, the notion of Feedback is deeply linked to the Observer, but the two are different concepts. In fact, the Feedback usually processes the information reported by one or more observers to decide if the execution is interesting. While the concept of "interesting" is abstract, it is typically related to a novelty search (i.e., interesting inputs are those that reach a previously unseen edge in the control flow graph). In another example [58], an Observer can be used to report all the sizes of memory allocations and a maximization Feedback can be used to maximize these values to spot pathological inputs in terms of memory consumption.

The process that identifies interesting inputs also has a second important goal in fuzzing: finding the *solutions* that satisfy specific objectives, for example, an observable crash in the target program. This type of feedback, the Objectives, act as oracle that describes the expected outcome of the fuzzing campaign, for instance, a set of crashing testcases with a unique stacktrace like in HonggFuzz or an input that triggers a crash along a specified path like in AFLGo [13].

Mutator – The Mutator is an entity that takes one or more Inputs and generates a new derived one. Mutators can be composed of other mutators and they are generally linked to a specific Input

type. In a traditional fuzzer, mutators are composed of many bit-level mutations like bit flip or blocks swapping. A mutator can also be informed about the input format and mutate the internal representation of the Input, for instance, by swapping nodes in an Abstract Syntax Tree in case of a grammar fuzzer. Mutators are usually the part of a fuzzer that changes more often when creating a custom fuzzer.

Generator – A Generator is a component designed to generate a new Input from scratch. For instance, a random generator can be used to generate random inputs. While less popular in feedback-driven fuzzing, there are notable exceptions that adopt Generators. For instance, Nautilus [4] uses a grammar-based generator to create the initial corpus and a sub-tree generator as a mutation of its grammar mutator.

4 FRAMEWORK ARCHITECTURE

The goal of LibAFL is to provide the basic blocks required to build a new generation of fuzzers through a modular design based on reusable components and reliable, fast, and scalable implementations of state-of-the-art techniques. To achieve this objective, we decided to bound the framework's design to the actual programming language that we use, Rust, by exploiting its features from the design stage. In this section, we present and discuss the design of LibAFL as a system and its individual components.

4.1 Principles and High-level Design

The LibAFL framework is designed around three key principles:

- **Extensibility**, to allow the user to swap different implementations of the entities explained in Sec. 3 in or out, without touching other parts. This allows the seamless combinations of orthogonal techniques but also ease the design and development of new components;
- **Portability**, most of the existing fuzzers are OS-specific, running either under *nix or Microsoft Windows. To avoid this pitfall, we opted instead to design our core library in a system-independent way. Moreover, for maximum portability, we implemented a subset of LibAFL, including all core components, without any dependency on any standard library, thus allowing the users to write fuzzers for bare-metal targets like embedded systems and kernels;
- **Scalability**, no design choices must conflict with the ability to scale fuzzers over multiple cores and/or machines. Because of this, we design an event-based interface that enables and facilitates the communication between fuzzers;

As we already discussed, none of the existing fuzzing frameworks are completely extensible. Some are portable on different operating systems, like LibFuzzer [46] but no one can compile on systems without a standard library. Last but not least, scalability is a known weakness of existing fuzzers. The design of AFL, and therefore of its many derivates, is based on disk IO communication and expensive syscalls such as fork(2) [75]. This causes a terrible performance when the fuzzer is scaled across multiple cores [23]. Other more scalable solutions, like HonggFuzz, are still based on syscalls to control the target and maintain a shared state between all the parallel threads that need locks contentions. On the other hand,
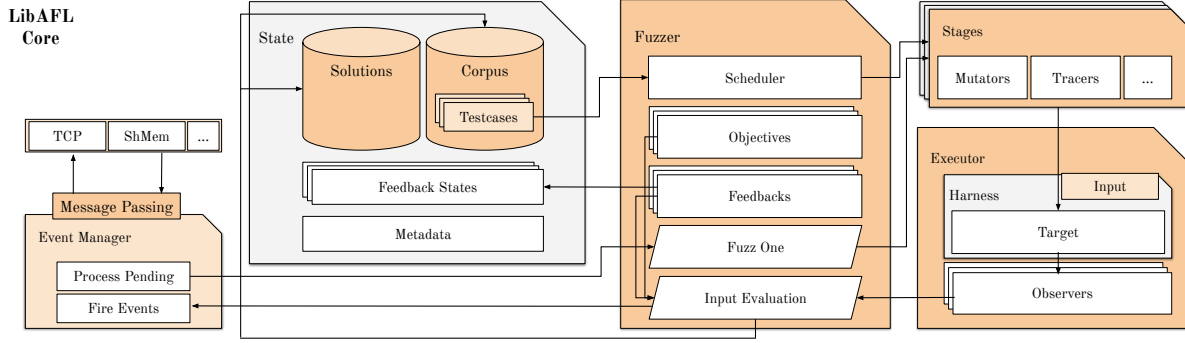
Figure 1: LIBAFL **Core architecture. Links are a representation of a non-comprehensive picture of the interactions.**

```
1  use libafl_sugar::InMemoryBytesCoverageSugar;
2  use libafl_targets::libfuzzer_test_one_input;
3
4  InMemoryBytesCoverageSugar::builder()
5      .input_dirs(input_dirs)
6      .output_dir(output_dir)
7      .cores(cores)
8      // For multi-node synchronization
9      .broker_port(broker_port)
10     .harness(|buf| {
11         libfuzzer_test_one_input(buf);
12     })
13     .build()
14     .run();
```

Listing 1: An example frontend with LIBAFL **Sugar.**

LIBFUZZER achieves greater scalability as different nodes cannot communicate while fuzzing, the corpus is merged after a defined time span, and the fuzzers are restarted.

To create a fuzzing framework following the three aforementioned objectives, we designed our system around three core libraries:

- **LibAFL Core** is the main library and contains the fuzzing components and their implementations. A large part of this library depends only on Rust core+alloc and so can run without any standard library;
- **LibAFL Targets** contains the code that lives within the target program, like the runtime library for coverage tracking;
- **LibAFL CC** provides the functionalities to write compiler wrappers for LIBAFL, by providing to the user a set of compiler extensions useful for instrumentation;

In addition to these three core libraries, LIBAFL contains several **Instrumentation Backends** that offer APIs to bridge LIBAFL to different execution engines, such as QEMU usermode and Frida.

In our naming convention, all these libraries are part of a toolkit that is used to create fuzzers called **Fuzzer Frontends**. Some ready-to-use frontends are already available in an additional library in the framework, LIBAFL Sugar, that provides a high-level glue API to quickly set up a frontend in just a few lines of code. We also provide Python bindings to the Sugar crate for quick prototyping without recompilation.

For instance, Listing 1 shows a simple fuzzer that bridges a LIBFUZZER-style harness to a fuzzer that uses generic bit-level mutations and executes the target in-process written using the high-level APIs of LIBAFL Sugar.

At the time of writing, the entire LIBAFL framework, tests included, consists of 53k lines of Rust and 15.4k lines of C/C++.

4.2 The Core Library

Figure 1 shows the architecture of the core library in terms of links between components. Most components are a one-to-one mapping with the entities we discussed in Section 3, with the addition of three additional macro-components: **State**, **Fuzzer** and **Events Manager**.

Each component is mapped to a Rust generic trait, allowing it to work in combination with any other orthogonal component. For instance, the State component can work with any type of Input, thanks to the fact that the definition of Input itself requires that any of its implementations must be serializable, and so usable in a corpus stored on disk.

This configuration of the architecture is the standard architecture for a frontend proposed in LIBAFL, but custom architectures can be defined too in which, for instance, there is no corpus, and the fuzzer is purely based on a generator and an executor. An alternative architecture, already in LIBAFL, consists of a pipeline without any executor in which it is not the fuzzer to feed the target program with generated inputs, but the fuzzer is a service from which inputs can be requested, allowing for instance to use LIBAFL in the context of hooks of some emulators such as Panda [20].

4.2.1 Zero-cost Abstractions. Extensibility comes with the price of introducing abstractions, which usually has a cost in terms of performance. As speed is an important metric in fuzzing, we devised a design that allows flexible abstractions without paying a noticeable cost at runtime.

Since the beginning, we avoided traditional object-oriented patterns to get rid of virtual tables and similar performance-killing approaches. The reason behind this choice, driven by micro-benchmarks performed during the early stage of the development, is not simply to avoid the cost of an indirect call, that is minimal, but to exploit the design of the Rust programming language in which all the code

is compiled alongside the libraries allowing powerful optimizations applied on heavily inlined code.

Instead, we designed all components as generic traits, each taking other related components as generic parameters. Sub-components are then defined via composition. In this way, we pay at compile time the cost of combinations of linked-but-independent entities, such an Executor and different kinds of Inputs. As a second design pattern, inspired by Haskell, we employ compile-time lists similar to hlist [56] to specify multiple objects, such as the set of observers of the set of mutations in a composable mutator. These lists have matching capabilities to retrieve single objects stored in the data structure. For instance, a feedback can access the observers that are useful to discriminate the interestingness of an execution by either name or type. By exploiting the powerful compile-time facilities offered by Rust our code is so compiler optimizations friendly and, as anticipated before, it can be aggressively inlined.

4.2.2 The State. The State is where all the non-volatile data resides. Everything that is part of the evolutionary algorithm's data must be included in the State, as well as the number of executions, the pseudo-random number generator state and the corpora (both the main and the solutions corpus).

As some types of feedbacks also need to maintain a state, for instance, the coverage observed so far in coverage-guided fuzzing, we introduce the **FeedbackState** component that is linked to both the state and the feedbacks. The instances of the feedback states are contained in state in a compile-time list and can be retrieved by name, typically by the linked feedback that wants to alter its state while deciding if an execution was interesting or not.

The main purpose of having a place for the data of the fuzzer is to exploit the serialization facilities of Rust. Serializing and de-serializing a state allows any LibAFL-based fuzzer to stop and later restart from the exact same internal state. For in-process fuzzing, this novel approach allows LibAFL to recover instances from crashes by re-loading a serialized state in the crash handler, without the need to re-execute the entire corpus as previous solutions do.

4.2.3 The Fuzzer. The Fuzzer is a recipient for the operations that define what the fuzzer can do. It contains the Feedbacks, the Objectives, and the Scheduler, all independent operations that may alter the fuzzer state. These stages are separated from the fuzzer to respect the borrowing rules of Rust [1], as they may invoke some operations that alter Fuzzer and State at the same time.

The Fuzzer, in addition, provides the definition of how a single testcase should be processed and how to evaluate a new input. By default, the standard implementation of it, which consists of feedback-driven fuzzing, defines FuzzOne, the operation responsible to process a single testcase, as the invocation of the scheduler to get the testcase to fuzz and the invocation of every stage on the testcase. InputEvaluation, the operation that evaluates if an input must be added to the corpus, is by default the execution of the target program and the decision if it is interesting or a solution using the feedbacks.

Custom architectures may need other structures implementing Fuzzer or State, allowing different designs like Vuzzer [62] that decouple input generation from the immediate evaluation to existence within LibAFL.

4.2.4 The Events Manager. The Events Manager is an interface for generating and processing events, which can be used to implement multi-node synchronization in a parallel fuzzer or simply for the purpose of logging. The Event Manager is designed to maximize scalability. In fact, if we assume a communication channel that scales linearly (such as shared memory message passing [67]) the Manager does not introduce any further bottleneck as each fuzzer works on completely separated data and the process of pending events is deferred to specific reconciliation points in the fuzzer loop, triggered before the fuzzer requests a new testcase to the scheduler.

Our framework includes a rich set of events. For instance, a component can be notified when one fuzzer adds a new testcase to its corpus, receiving an event containing a serialized version of the input and the set of observers that were considered interesting by the feedback.

4.2.5 The Metadata System. Fuzzing algorithms often need to reason about the information associated with a given testcase or the overall state of the fuzzer. Therefore, LibAFL must provide a way to extend the data in the testcases of the corpus and the data maintained in the state. A naive but effective solution would be to redefine new types by composition, but in this case, the developer would need to be aware of each piece of metadata required by all the employed algorithms. Thus, in order to provide this capability while maintaining simplicity and performance, we designed a dedicated Metadata System for the State and Testcase components. In particular, in LibAFL any struct that implements the SerdeAny trait, a trait that we created to allow the serialization of trait objects [2] without the requirement of a standard library, can be used as metadata. This trait so requires serialization capabilities and static lifetime [3] as the instances must be able to be converted to trait objects.

LibAFL provides then serializable maps that can store any instance that can be cast to a SerdeAny trait object. Both the Testcase and State holds a map of this type as an extensible container for metadata. In this way, different but related components can cooperate by operating on the same metadata, while completely ignoring what the other components do with their own metadata. However, this is the only pattern in LibAFL that introduces a small runtime overhead, due to the map lookup (currently implemented as a hashmap).

4.2.6 Composable Feedbacks. A fuzzer may require combining multiple feedbacks to evaluate how "interesting" was a given input or to support different objectives. In LibAFL, to avoid the need to create new aggregated feedbacks from scratch, feedbacks can be composed by using logical operators. For instance, a fuzzer may not want to save every crashing input but instead perform some sort of crash de-duplication. In LibAFL, this can be achieved for example by using a feedback that considers crashing inputs as interesting and one that considers an input interesting when it

---

[1]https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[2]https://doc.rust-lang.org/book/ch17-02-trait-objects.html
[3]https://doc.rust-lang.org/rust-by-example/scope/lifetime/static_lifetime.html

LibAFL: A Framework to Build Modular and Reusable Fuzzers

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

triggers a new stack trace never observed before. In this case, a crash de-duplication objective can be achieved simply by putting the two aforementioned feedbacks in the AND.

4.2.7 The Monitor. The last component in a LIBAFL-based fuzzer is the **Monitor**. It is the component that maintains the statistics collected from the triggered events and displays them to the user. While this component is not fundamental for a working fuzzer, the lack of human introspection reduces the effectiveness of a fuzzing campaign. Monitors allow the developers to report and display custom stats and to implement various reporting interfaces, such as printing a status screens in the terminal or forwarding the data to a Grafana web interface with statsd [4].

4.3 Instrumentation Backends

LIBAFL can be easily plugged into any instrumentation backend, like a binary translator or a simple compiler instrumentation pass. By default, we provides additional libraries that tie LIBAFL with some popular instrumentation backends: LLVM [42], SanitizerCoverage [44], QEMU usermode [9] and Frida [1].

The runtime in LIBAFL Targets can be linked to any SanitizerCoverage target having so coverage and comparisons tracking in the fuzzer with the addition of a compiler flag. The SanCov support allows the users to create frontends compatible with any SanCov-enabled target, not only C/C++, such as Atheris [32] for Python and cargo-fuzz [8] for Rust.

LIBAFL CC provides a set of LLVM passes to extend Clang and other LLVM-based compilers to track edge coverage, context-sensitive and K-context-sensitive [7] edge coverage, N-gram coverage [71], coverage accounting [73], comparisons with CmpLog [6, 27] and dictionary tokens with autotokens, and an enhanced version of the AFL++'s `dict2file` pass that extracts the tokens from interesting functions such as `strcmp`.

LIBAFL QEMU bridges QEMU usermode, and full system in the near future, to Rust with a novel emulator API with hooking capabilities to have programmatic control over the PUT execution. Around this interface to QEMU, the library exposes structures like executors and helpers to install default hooks for common fuzzing tasks, such as edge coverage tracking, guest snapshot-restore, and binary-only ASan [26].

LIBAFL Frida offers similar capabilities to the QEMU bridge, but with the features of a DBI and so without the host-guest separation. It includes a binary-only ASan and, unlike QEMU usermode, it can work on various operating systems other than Linux such as Windows, macOS, and Android.

Instrumentation capabilities in LIBAFL are also offered for concolic execution, through LIBAFL Concolic and its Rust bridges to SYMCC [60] and SYMQEMU [61]. Our API allows a user to write custom constraint collection filtering in Rust and to report to a LIBAFL-based fuzzer constraints in an easy-to-manipulate format. These constraints can then be used in a mutator, for instance, to generate inputs invoking a solver, or fuzzed as Borzacchiello et al. did in their recent work [15].

---

⁴https://github.com/statsd/statsd

Table 1: List of techniques integrated in LIBAFL, **the first part only contains the techniques evaluated in Section 5.**

Technique	New components	Additional instrumentation
REDQUEEN [6]	✓	✓
Auto-tokens [27]		✓
Value-profile [47, 58]		✓
Block coverage accounting [73]	✓	✓
Function coverage accounting [73]	✓	✓
Loops coverage accounting [73]	✓	✓
Corpus culling scheduler [76]	✓	
Weigthed scheduler [27]	✓	
FAST power schedule [14, 27]	✓	
COE power schedule [14, 27]	✓	
EXPLORE power schedule [14, 27]	✓	
MOPT [48]	✓	
NAUTILUS [4]	✓	
GRIMOIRE [10]	✓	
GRAMATRON [68]	✓	
Token-level [63]	✓	
NeoDiff [50]	✓	
LIN power schedule [14]	✓	
QUAD power schedule [14]	✓	
EXPLOIT power schedule [14]	✓	
SYMCC [60]	✓	✓
SYMQEMU [61]	✓	✓
Hitcounts [76]	✓	✓
Ngram coverage [71]		✓
Context-sensitive coverage [17]		✓
QASan [26]		✓
Atheris (compatibility) [32]	✓	
cargo-fuzz (compatibility) [8]	✓	

In addition to these stable backends, LIBAFL has already a partial support to TinyInst [28] to instrument binaries on Windows and macOS and NYX [64] for hypervisor-level snapshot fuzzing.

5 APPLICATIONS AND EXPERIMENTS

In this section, we discuss some of the techniques implemented in LIBAFL and their relation with the entities presented in the previous sections. While LIBAFL already has many implemented techniques, in the first part of this section, we focus on four popular problems in fuzzing that are the focus of many published works in the literature: roadblocks bypassing (e.g., [6, 17, 57, 62]), structure-aware fuzzing (e.g., [4, 10, 25, 59]), corpus scheduling (e.g., [18, 21, 72, 73]) and energy assignment (e.g., [12–14, 43]).

A non comprehensive list of the techniques integrated in LIBAFL at the time of writing is listed in Table 1, alongside the information if the technique require additional implementations of components in the fuzzer side or additional instrumentation code in the target side.

After discussing how these techniques are implemented, we evaluate them in three different sets of experiments:

(1) We measure the performance in terms of code coverage and bug detection of several approaches implemented and ready-to-use in LIBAFL;

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti

(2) We show how we can combine orthogonal approaches implemented in our framework to build new and never evaluated before fuzzers and measure their performance;

(3) Lastly, to show the efficiency of our framework in a traditional context, we compare and evaluate a new generic bit-level fuzzer based on LibAFL using the previously presented techniques against other state-of-the-art fuzzers like AFL++ and HoNggFuzz on FuzzBench [54];

We ran the first two sets of experiments on a x86_64 machine equipped with an Intel® Xeon® Platinum 8260 CPU with a clock of 2.40 GHz. The dataset is a subset of the FuzzBench suite, selected to include programs with diverse features. We run each session for 24 hours and repeated each experiment five times to mitigate the effect of randomness in fuzzing.

For the comparison with other fuzzers, we run the experiments on the full FuzzBench suite on the service offered by Google. Each run was 23 hours long and repeated 20 times. The benchmarks suite provides the initial corpus for every benchmark and we used the default setting. The overall ranking that we discuss is based on the average normalized score computed by FuzzBench, which represents the percentage of the highest reached median code or bugs coverage on a given benchmark.

Finally, in the last part of the section, we present a case study about how easy, we can implement with LibAFL a fuzzer that is different from the traditional setups like the ones shown in the first part, a differential fuzzer that can spot logic bugs in Ethereum VMs with a different kind of feedback based on the state of the VM instead of code coverage.

In the end, we discuss other implemented approaches and their relations without providing an evaluation for the sake of time and brevity.

5.1 Roadblocks Bypass

An important field of research in fuzz testing is the development of new techniques to increase code coverage by bypassing hard-to-solve constraints. For instance, multi-byte comparisons pose a severe problem for fuzzers that employ a generic bit-level mutator, since random generic mutations cannot bypass these comparisons as the solution space is huge and blind guessing impractical. LibAFL provides several ready-to-use techniques to implement fuzzers that can overcome these roadblocks.

The first, in chronological order of appearance in the literature, is the *value-profile* [45] proposed by LibFuzzer in 2016. This technique tries to solve comparison instructions by maximizing the number of matching bits between the two operands of the instruction. In LibAFL, this is implemented with a map observer and a feedback that maximizes the entries of a map and considers an input as interesting when at least one new max is discovered. This type of feedback, MaxMapFeedback, is builtin into our framework and can be easily combined in OR with the basic edge coverage.

The second technique provided by LibAFL is *cmplog* from AFL++. This solution is based on the approach adopted by RedQueen [6] and Weizz [25], and can bypass comparisons by finding and replacing input-to-state values. It works by instrumenting the comparison instructions and any function with two pointers as arguments, and

Table 2: Uncovered code coverage over time (24h) of the roadblocks bypassing experiment.
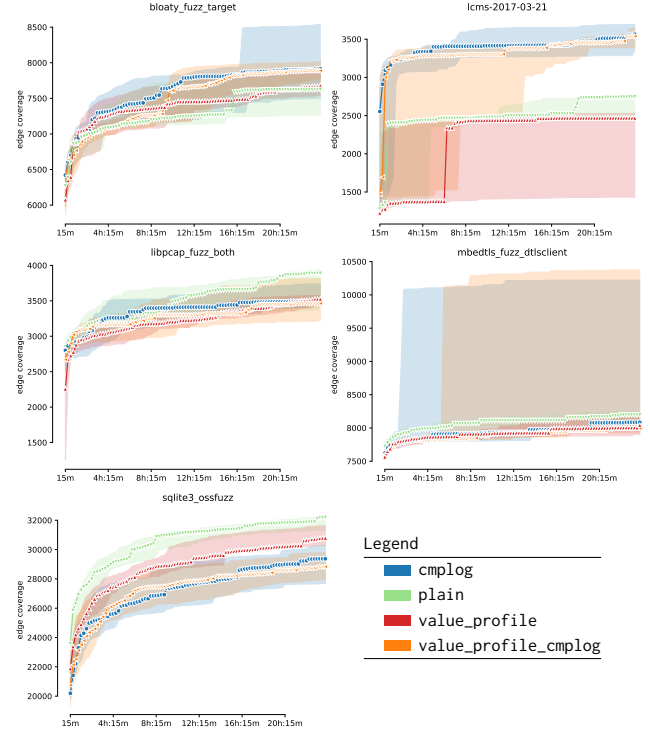
logging the related values in a map at runtime. This logging operation is done only once for each testcase in the corpus and in our framework this is implemented with a second executor with one observer that handles the cmplog map. The executor is invoked in a tracing stage at the beginning of the pipeline and the logged values are stored as metadata. Later on, a custom mutator matches the pattern in the input and replaces them with the other operand of the comparison in a specific mutator.

The third technique, *autotokens*, was also inspired by AFL++, where it can be used only when instrumenting the target with an LTO pass. In LibAFL CC, this instrumentation is available also for the regular compilation. This pass extracts the tokens from the comparison instructions and the functions with immediate values, encoding them in a section of the binary. A LibAFL-based fuzzer can then retrieve the tokens, add them to the dictionary in the State's metadata, and use the tokens in the mutator without any overhead. For this experiment, we consider autotokens as the baseline. In fact, since it does not introduce any overhead, there is no reason to not use it in a fuzzer.

Cmplog and value-profile, on the other hand, require additional instrumentation and value-profile can bloat the corpus with more input as it increases the sensitivity [71] of the fuzzer. Thus, we now evaluate four different options. These include plain (the baseline with autotokens), value_profile, and cmplog, as well as a new value_profile_cmplog which combines both the aforementioned

LibAFL: A Framework to Build Modular and Reusable Fuzzers

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

Table 3: Uncovered bugs after 24h of the structure-aware fuzzing experiment.

Table 4: Uncovered bugs after 24h of the Nautilus +MOpt **fuzzing experiment.**
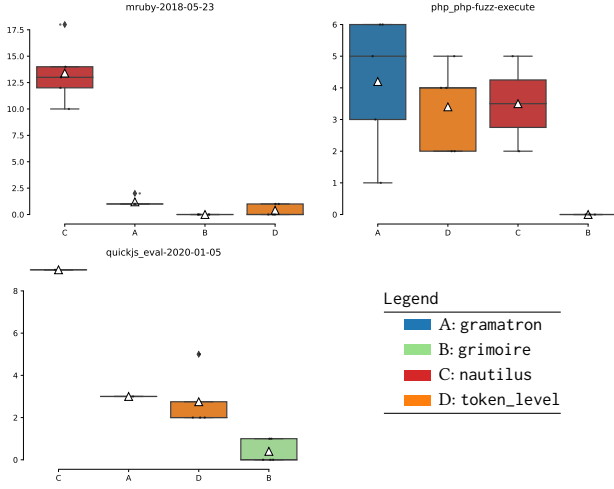
techniques. This combination was never evaluated in previous studies and shows how the composability of our solution allows experimenting with different combinations of components and simplifies the development of complex fuzzers.

Table 2 reports the coverage growth graphs over 5 benchmarks from FuzzBench. Overall, `cmplog` is the best performer (95.94), closely followed by `value_profile_cmplog` (95.03), and `plain` (94.65). Instead, `value_profile` performed considerably worse (90.13). This is interesting, as it suggests that autotokens alone is able to solve many roadblocks without additional overhead, allowing `plain` to shine on libpcap which is a benchmark with many input-to-state comparisons.

This confirms that most roadblocks are input-to-state and the solving capabilities of `value_profile` are not an adequate reward for the additional sensitivity it introduces (due to the possible internal wastage of the corpus by too many similar testcases). We think that the combination of the two techniques, however, can have interesting target-specific applications that can be investigated in the future.

5.2 Structure-aware Fuzzing

While generic mutators can effectively stress parsers by generating invalid inputs, it is also important to fuzz deeper paths beyond the parsing routines to spot bugs in these code regions. A common solution to this problem is to make the fuzzer aware of the input format. While generating testcases from a specification is one of the oldest embodiments of fuzz testing, recent works in the literature have explored the combination of modern feedback-based fuzzing with a mutator that is structure-aware [4, 33, 59]. Beyond this, other approaches [10, 25, 63] proposed to approximate structure-aware fuzzing with learning heuristics without requiring any user-provided specification, thus working on targets without a known input format and reducing the amount of human work.

LibAFL provides several techniques to deal with structured inputs, taking advantage of the flexibility of all the other components
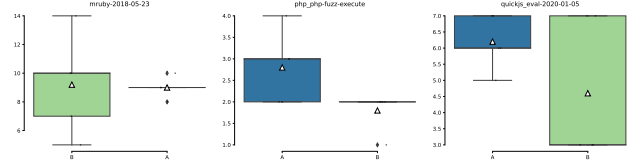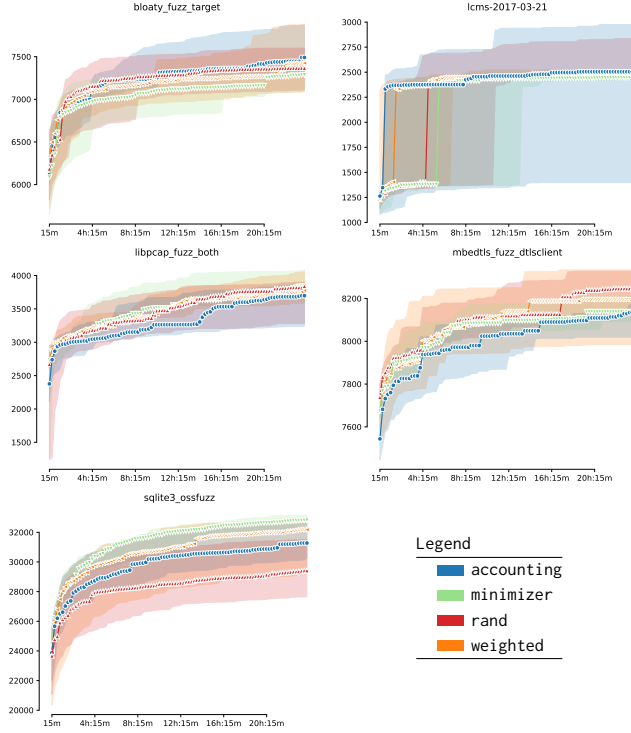
that are agnostic to the input type. Nautilus [4] is a grammar-based coverage-guided fuzzer that evolves a corpus of syntax trees with mutations like subtree generation and replacement from another input in the corpus. In our framework, we implemented an input type for Nautilus, a generator that can create testcases from scratch, and a set of mutators that make use of the generator to create subtrees. All the other entities remain untouched and immediately compatible with the grammar fuzzer. For instance, the `ScheduledMutator` (the trait for mutators that can schedule other mutators as mutations), is employed seamlessly with a maximum of 8 stacked mutations. Another technique available in our framework is a re-implementation of Gramatron [68], a grammar-based fuzzer that employs a grammar-to-automata conversion to implement fast mutators. In LibAFL the grammar preprocessing utility is provided as a tool and the associated structures are specular to the ones used by Nautilus, with a different underlying implementation.

As an example of approaches that perform grammar learning, LibAFL implements Grimoire [10], a fuzzer that uses the portion of inputs that induced the novelty in coverage as tokens to build generalized "tree-like" inputs and perform grammar-like mutations. It also employs token-level fuzzing [63], an approach based on token extraction with a lexer. While the original solution was specific to JavaScript, our implementation is generic and can be applied to any programming language. Classic bit level mutations are then applied to encoded inputs and that is decoded before the target execution.

To evaluate these three approaches we decided to use the number of uncovered bugs instead of code coverage. In fact, the effectiveness of this type of fuzzers is less dependent on code coverage, especially when comparing with variants that can generate invalid inputs and so inflate the coverage by exploring error paths.

In this experiment, as both Grimoire and token-level require some initial seeds, we used Nautilus to generate 4096 initial inputs for these two fuzzers to avoid a bias due to the quality of the seed corpora [37] provided by FuzzBench. Table 3 shows the number of bugs found by each variant over 3 popular compilers. Grammar-aware approaches are still superior, with a huge boost for Nautilus on 2 benchmarks over 3, but the performance of the token-level approach is surprisingly similar to Nautilus on PHP, given that they start from similar seed inputs.

Given the promising results of this experiment, we decided to investigate whether we can further improve the best performer, Nautilus, by combining it with other orthogonal techniques provided by LibAFL. For instance, we combine it with the MOpt mutation scheduler [48], to create a new and never evaluated variant. MOpt has been used to date only to schedule bit-level mutations by assigning probabilities to mutations based on the observed effectiveness

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti

Table 5: Uncovered code coverage over time (24h) of the corpus scheduling experiment.

during a learning phase by using a Particle Swarm Optimization algorithm. We repeated the experiments and compared this new variant against the 3 grammar benchmarks, running the fuzzer 5 times for 24 hours. The results are reported in Table 4 that shows how Nautilus (▬) perform well when coupled with MOpt (▬) on mruby, but worst on php. Overall, this confirm the highly target-dependent nature of MOpt that was observed [27] for bit-level fuzzing.

5.3 Corpus Scheduling

The choice of the next testcase to use from the corpus is the focus of many different studies. The simplest solutions rely either on random selection or on a FIFO queue. LibAFL provides both, along with additional schedulers inspired by other state-of-the-art fuzzers.

The first is taken from AFL, which in every queue cycle selects a subset of "favored" seeds from the corpus. The seeds are chosen based on execution speed and input length while preserving the maximum coverage. In LibAFL, we implemented a generic version of this approach called `MinimizerScheduler`, which computes the minset based on the entries of a given map feedback but with customizable weights. For simplicity, in the following experiment, we use the traditional weighting policy used by AFL.

The second scheduler uses a recent improvement proposed in AFL++ based on probabilistic sampling. The idea is to map each testcase in the corpus to a probability and a more "promising" neighboring testcase in terms of a computed score. The score is computed by using various metrics, including execution time and coverage map

size, and it is used to calculate the probability too. The selection process consists in choosing a random testcase from the corpus and a random number between 0 and 1. If this number is less than the probability associated with the testcase, this testcase is selected, otherwise, the more promising neighbor is picked.

The third scheduler is taken from TortoiseFuzz [73] and prioritizes inputs by using three security impact metrics: memory operations with both block and function granularity, and loop back edges counting. The tracking of these metrics requires a custom instrumentation, implemented in LibAFL CC with LLVM and a new associated observer. The scheduler prioritizes inputs with higher scores and, breaking ties based on input length and execution time.

Table 5 shows the results of four fuzzers based on the aforementioned schedulers: `accouting` is the fuzzer using the TortoiseFuzz instrumentation and a scheduler with function-level granularity, `minimizer` is using the AFL's algorithm, `rand` the random selection baseline, and `weighted` is using the probabilistic scheduler from AFL++.
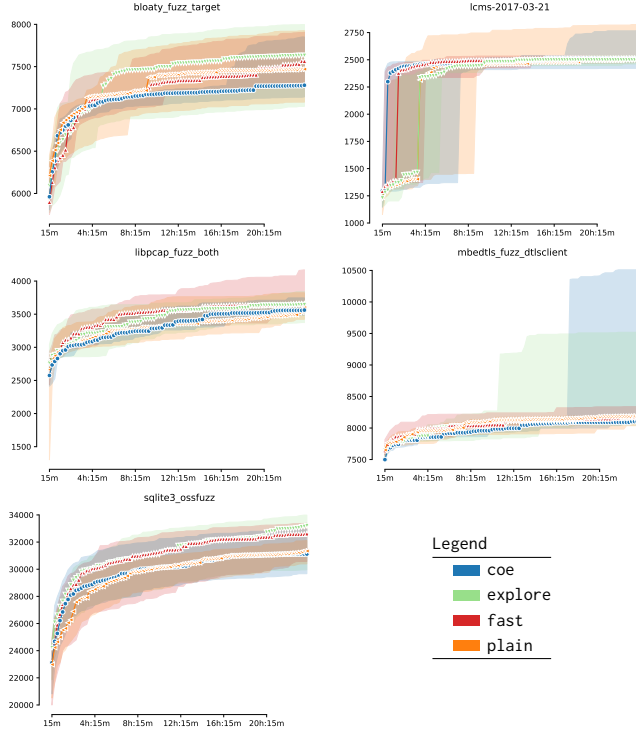
In terms of average normalized score of the uncovered coverage, `weighted` achieves the best results (with a score of 98.91), closely followed by `minimizer` (98.71), `accounting` (98.03), and `rand` in last position (97.50). The small difference among all solutions is interesting and shows that, despite the huge attention given to this problem in the literature, a simple random approach still achieves decent results and it is perfectly suitable for real-world fuzzing campaigns on fast targets. We also expected `accouting` to outperform `minimizer`, which instead was not the case. However, unlike in the original evaluation carried out in the TortoiseFuzz paper by using AFL, LibAFL achieves much higher throughput by executing the target in-process, thus reducing the difference and impact of these scheduling techniques. While on fast targets the difference is minimal, we believe that the scheduling problem is crucial on slow targets, where deciding which testcase to fuzz beforehand can have a large impact on the fuzzing campaign.

5.4 Energy Assignment

Energy assignment tries to answer the question of how many times a single input in the corpus needs to be mutated to create new testcases. The general problem, also known as *power scheduling* problem, was introduced in the literature by Böhme et al. [14] in 2016.

The most naive solution is to use a constant value, while the most commonly used simple approach [47, 69] assigns to each seed a random value in a given interval. LibAFL also provides this simple algorithm, named `plain`, with a range between 1 and 128 when using the default mutational stage.

While many solutions tune this specific parameter, often for domain-specific fuzzers [13, 59], the seminal work of AFLFast remains the most complete coverage of this problem for generic fuzzing. AFLFast proposed six different algorithms: *exploit*, *explore*, *coe,fast*, *lin* and *quad*. In detail, *exploit* assigns energy proportional to some metrics like execution time of the input, coverage density, and creation time of the testcase. *Explore* assigns low energy, dividing the energy of exploit by a constant. *Coe* is an exponential scheme that assigns 0 as energy to inputs that trigger high-frequency edges until they become low-frequency. *Fast* is an extension of *coe*, in

LibAFL: A Framework to Build Modular and Reusable Fuzzers

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

Table 6: Uncovered code coverage over time (24h) of the energy assignment experiment.

which instead of assigning 0 the scheme assigns energy inversely proportional to the amount of visited high-frequency edges. *Lin* assigns energy linearly w.r.t the times the testcase has been chosen to be fuzzed, while *quad* is quadratic.

In LibAFL, we designed an interface for power scheduling based on metadata, on top of this, we implemented the six aforementioned algorithms. Our implementation is, however, based on an optimized version of the algorithms integrated into AFL++, which was developed years after the AFLFAST paper and that was never evaluated in the literature. Among the six, the most effective[5] are *explore*, the default in AFL, and *coe* and *fast*, the default in AFL++. Therefore, we will focus our tests on these three algorithms and compare them with a baseline (never evaluated before) based on the `plain` algorithm. Table 6 shows the results in terms of code coverage.

Overall, considering the average normalized score across all benchmarks, `explore` is the best performer (99.72), with `fast` following (99.43) and then `plain` (98.12) and `coe` (97.08). This results confirm the trend observed in the AFL++ version of the power schedules, with *fast* and *explore* as top performer, with *fast* as the now default schedule in AFL++.

The LibAFL implementations emphasize once again the same observations we did for the corpus scheduling problem. Fast fuzzers (on fast targets) reduce the benefit that can be gained by using more complex scheduling as less useful executions have a limited effect

on the throughput. The score of `coe`, which performs worse than the random baseline, can be considered target-specific. In fact, while the algorithm suffered on some targets (particularly on the `bloaty` benchmark), it was the best performer on others (e.g., on a few runs of `mbedtls`).

5.5 A Generic Bit-level Fuzzer

While the main goal of LibAFL is to be a Swiss-army knife to build custom fuzzers, we also strive to provide good default implementations to use for out-of-the-box generic bit-level fuzzing.

In this section, we present a frontend for LibAFL to fuzz LibFuzzer harnesses with a generic mutator. We evaluate this fuzzer against the state-of-the-art fuzzers used in Google OSS-Fuzz to fuzz thousands of open source projects every day, AFL++, HonggFuzz, and LibFuzzer, with the Entropic [12] option enabled for better performance.

Our fuzzer, like LibFuzzer and any other fuzzer used in the previous experiments, uses an in-process executor to run the target harness, while AFL++ and HonggFuzz control the target with forms of IPC (like pipes). Our fuzzer also employs some of the improvements we covered in the previous experiments: cmplog, the weighted corpus scheduler, the *explore* energy assignment scheme, and the MOpt mutator.

For this particular experiment, we submitted a request to the FuzzBench service [6], to execute the four fuzzers on 22 different benchmarks and evaluate the reached code coverage. Each fuzzer was scheduled for 23h and every single experiment was repeated 20 times to mitigate the randomness. The results for each benchmark are reported in Table 7. The overall result, based on the average normalized score of the coverage uncovered across all the targets, is that LibAFL clearly outperforms all the other fuzzers with a score of 98.61, followed by HonggFuzz (96.65), AFL++ (96.32) and Entropic (94.22). This result is even more relevant since the other fuzzers were gradually improved over time [54] based on the outcome of several FuzzBench runs, while LibAFL has been developed independently from this benchmarking suite.
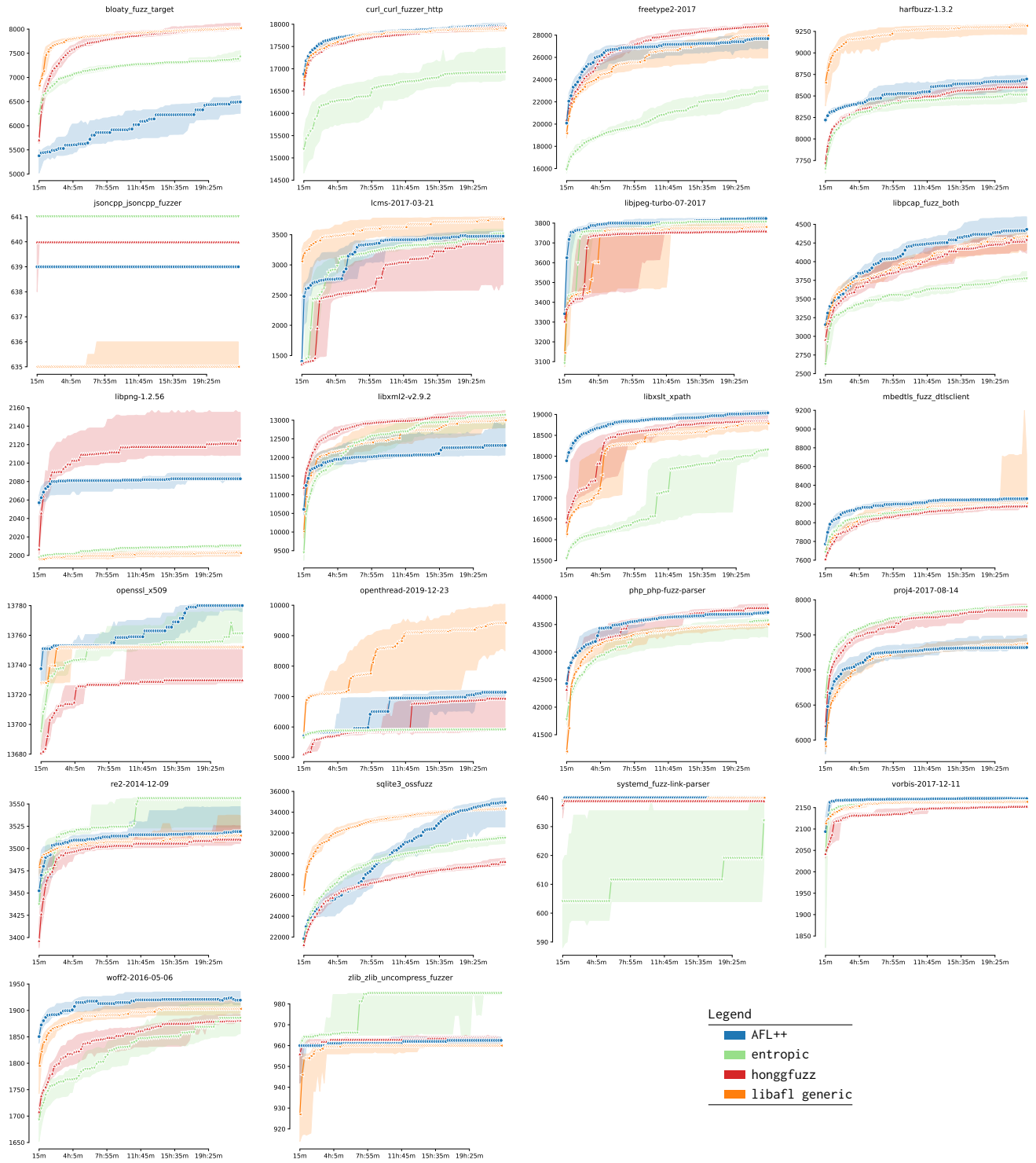
By inspecting the results in isolation, we can see that LibAFL shines on 3 benchmarks, `harbfuzz`, `openthread` and `sqlite3`. Notably, it is the only fuzzer that can reach the coverage breakthrough by unlocking the fuzzer from saturation in a few runs on `mbedtls` and consistently on `openthread`.

On the other hand, our fuzzer clearly underperforms AFL++ and HonggFuzz on `libpng`, resulting in an almost equal performance with Entropic. The missing coverage of these two fuzzers can be explained with the execution model that they use, in-process, versus the out-of-process executor that the other two uses. The latter is slower but more reliable in handling timeouts. The strength of our approach is that limitations like this one can be easily overcome by changing a few lines of code in the frontend, in this particular case to move from `InProcessExecutor` to `ForkserverExecutor`.

Overall, these results show that LibAFL is a mature framework capable to be the backbone of a modern generic fuzzer that can compete with state-of-the-art solutions. We foresee the development

---

[5]https://github.com/google/fuzzbench/issues/249#issuecomment-700470906

[6]The complete details of the experiment are available at https://www.fuzzbench.com/reports/experimental/2022-04-11-libafl/index.html

Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti

Table 7: Uncovered code coverage over time (23h) of the generic bit-level fuzzer experiment.

LibAFL: A Framework to Build Modular and Reusable Fuzzers

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

of a new version of highly customizable but with good defaults generic implementations, fuzzers (like AFL++) based on LibAFL.

5.6 Differential Fuzzing

In the previous sections, we presented variants based on coverage-guided fuzzing. However, LibAFL is not limited to code coverage and to its derivates (like context and ngrams [71]), but can also work with other types of feedback. As an example, in this section, we discuss the development of a frontend, inspired by NeoDiff [50], for differential fuzzing of two Ethereum virtual machines.

In short, NeoDiff, originally written in Python, compares the outcome of the executions of two VMs provided with the same input. To do so, it uses a *state hash*, a hash of the registers values, memory, and a probabilistic sampling of the stack at each instruction site of the executed trace. As feedback to evolve its corpus, it uses instead a *type hash*, the hash of the opcodes and the types of the first two items on the stack for each instruction. Any input that generates a trace with a new type hash is added to the corpus.

In LibAFL, NeoDiff can be implemented by taking advantage of the differential executor component, a structure that acts as a proxy to two underlying executors. The type of the inner executors is CommandExecutor, a simple kind of executor that spawns a new process given a command line, and its observers are linked to the stdout of such commands, as the Ethereum VMs print their traces in JSON after executing the input bytecode. These observers are processed by a custom feedback, the TypeHashFeedback that decodes the trace, computes the type hash, and compares it to the other hashes observed so far in the linked feedback state. A differential feedback, the feedback responsible to compare two observers, is used as objective. The differential feedback uses the state hash to compare the observers linked to the two executors and, if different, considers the input as solution only if having a novel typehash for de-duplication.

Overall, re-implementing NeoDiff from scratch in LibAFL took 2 working days person and consists of 900 lines of Rust code. We also decided to compare it against the original implementation of NeoDiff, by using the same metric adopted in the original paper, i.e., the number of diffing inputs with unique typehash. We run both fuzzers for 12 hours testing the same go-ethereum and openethereum versions tested in the original paper. In Figure 2 we report the findings over time of both fuzzers, showing that our implementation clearly outperforms the original in this metric. We believe that the bit-level mutators built-in in LibAFL play a major contribution in this experiment.

In terms of unique diffing instructions, the original paper reports that 6 instructions are the causes of the found differences. During our experiment, we reproduced the NeoDiff results finding only 5 instructions over 12h with NeoDiff and 15 instructions with our implementation.

The diffing opcodes in details are (in hex): 3, 31, 38, **3b**, 3c, **3f**, 44, **45**, **46**, 52, 5a, f1, f2, f4, fa. The opcodes found by both the original and our NeoDiff implementation are in bold, the others were discovered only by the LibAFL-based variant.

Our findings are so a superset of the 6 NeoDiff instructions, showing that LibAFL is outperforming the python implementation by a huge margin.
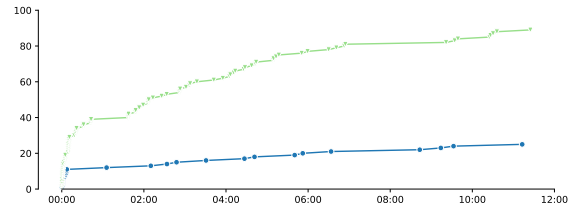

Figure 2: Uncovered diffing inputs (unique type hashes) for the original NeoDiff (■) and the LibAFL version (■) over 12h.

5.7 Third-Party Applications

During its development, LibAFL had already been adopted to implement several fuzzing frontends by a number of new users, who had no previous experience with our framework. For instance, it has been used to create a symbolic-model-guided fuzzer, tlspuffin [3] that employs a concrete semantic to execute TLS symbolic traces, thus proposing a new approach that mixes fuzzing and model testing. Thanks to this combination, tlspuffin can reach critical protocol states that are impossible to find with classic coverage-guided fuzzing.

A second third-party application of LibAFL is a snapshot fuzzer based on KVM, Tartiflette [19], which provides a new executor to run a Linux ELF as a VM with system calls emulation and instrumentation facilities for coverage tracking and snapshotting.

Finally, another LibAFL-based project worth mentioning is bananafzz [38], a fuzzer to detect race conditions with a novel design based on loop-per-thread calls generations.

6 LIMITATIONS AND FUTURE WORK

While extensible by design, the current implementation of LibAFL still lacks some components that are required to implement some specific fuzzing applications.

For instance, at the time of writing LibAFL CC does not include Link Time Optimization passes to reason about the whole program Control Flow Graph. This type of instrumentation is required to implement most of the directed fuzzing approaches [13, 16, 57] and thus, LibAFL is not currently providing any directed fuzzing application. This limitation, however, is not intrinsic to our design and support for directed fuzzing will be integrated in the near future.

A powerful feature integrated into LibAFL is the concolic tracing API, which can be used to extend SymCC or SymQEMU with custom constraints filtering and communicate the symbolic trace to a LibAFL-based fuzzer. Currently, LibAFL provides a solver stage based on Z3 that generates new testcases as traditional concolic fuzzers do. However, there are two main limitations in traditional concolic fuzzers that our architecture could help to overcome. First, solvers are hard to scale and are both time- and resource-consuming tasks. This could be mitigated by solving symbolic expressions [15] using fuzzing techniques [6, 17, 25]. The other limitation is that fuzzers and concolic engines poorly cooperate. Even when a solver outputs a testcase that solves a complex expression, it is very hard for a generic bit-level fuzzer to mutate and stress the program points

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti

related to this testcase without breaking the validity of the solved expressions. Approaches like Pangolin [40] go in this direction.

The possibility to build mutators using concolic expressions in LibAFL allows developers to implement these approaches, to overcome the previously mentioned limitations and reproduce previous experiments (e.g., the Pangolin artifacts have never been publicly released) However, none of these have been currently implemented in LibAFL.

Finally, a core principle of LibAFL is scalability. Therefore, an interesting future work would be to evaluate different fuzzing synchronization approaches in terms of scalability. LibAFL already implements an event manager capable of, if the target permits, scale linearly over multiple cores and machines. It also provides an alternative AFL-like disk-based method to synchronize testcases over nodes. An interesting research question is to measure how different approaches like TCP connections or shared memory-based communication affect fuzzing and to pinpoint their trade-offs.

7 CONCLUSION

In this paper, we presented a novel and completely extensible fuzzing framework, LibAFL. To show its versatility, and the comprehensiveness of its ready-to-use components to build state-of-the-art fuzzers, we presented several frontends based on LibAFL and performed experiments with them covering different problems in the fuzzing literature. We highlighted the customization allowed by the LibAFL design and the power of the combination of several orthogonal techniques, leading to a fuzzer that outperforms the best publicly available tools.

FOSS and Open Science. LibAFL is Free and Open Source Software under the license Apache 2.0 or MIT, at user option. It is available online at https://github.com/AFLplusplus/LibAFL. To allow the reproduction of our results and promote Open Science, every frontend developed for the experiments in this paper and the related setup to run them in FuzzBench is available online at https://github.com/AFLplusplus/libafl_paper_artifacts.

ACKNOWLEDGEMENTS

REFERENCES

[1] [n.d.]. Frida - A world-class dynamic instrumentation framework. https://www.frida.re/. [Online; accessed 10 April. 2022].
[2] [n.d.]. Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz. [Online; accessed 10 April. 2022].
[3] Maximilian Ammann. 2021. *Symbolic-Model-Guided Fuzzing of Cryptographic Protocols*. Master's thesis. University of Augsburg, Institute for Software & Systems Engineering. *See https://github.com/tlspuffin/tlspuffin*.
[4] Cornelius Aschermann, Tommaso Frassetto, T. Holz, Patrick Jauernig, A. Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS*.
[5] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*.
[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/
[7] Giorgio Ausiello, Camil Demetrescu, Irene Finocchi, and Donatella Firmani. 2012. K-Calling Context Profiling. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 867–878. https://doi.org/10.1145/2384616.2384679
[8] Rust Fuzzing Authority. [n.d.]. cargo-fuzz. https://github.com/rust-fuzz/cargo-fuzz. [Online; accessed 10 April. 2022].
[9] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401
[10] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko
[11] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 235–252. https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko
[12] Marcel Böhme, Valentin Manès, and Sang Kil Cha. 2020. 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)*. 1–11.
[13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020
[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[15] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. https://doi.org/10.1109/ICSE43902.2021.00071
[16] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2095–2108. https://doi.org/10.1145/3243734.3243849
[17] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046
[18] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 77–92.
[19] Tanguy Dubroca César Belley. 2021. Tartiflette: Snapshot fuzzing with KVM and libAFL. https://www.lse.epita.fr/lse-winter-days-2021/slides/lse_winter_days_tartiflette.pdf. [Online; accessed 10 April. 2022].
[20] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan Zee (north) bridge: mining memory accesses for introspection. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM, 839–850. https://doi.org/10.1145/2508859.2516697

LibAFL: A Framework to Build Modular and Reusable Fuzzers

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

[21] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 60–71. https://doi.org/10.1109/ICSE.2019.00024

[22] M. Eddington. [n.d.]. Peach fuzzing platform. https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html. [Online; accessed 10 April. 2022].

[23] Brandon Falk. 2020. aflbench. https://github.com/gamozolabs/aflbench. [Online; accessed 20 Dec. 2021].

[24] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2829–2846. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[25] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. 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. https://doi.org/10.1145/3395363.3397372

[26] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*.

[27] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[28] Ivan Fratric. [n.d.]. TinyInst. https://github.com/googleprojectzero/TinyInst. [Online; accessed 10 April. 2022].

[29] Patrice Godefroid. 2007. 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)*. 1–1.

[30] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08)*.

[31] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 50–59. http://dl.acm.org/citation.cfm?id=3155562.3155573

[32] Google. [n.d.]. Atheris: A Coverage-Guided, Native Python Fuzzer. https://github.com/google/atheris. [Online; accessed 10 April. 2022].

[33] Google. [n.d.]. libprotobuf-mutator. https://github.com/google/libprotobuf-mutator. [Online; accessed 10 April. 2022].

[34] Samuel Groß. 2018. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines pdfsubject=Not set.

[35] Richard Hamlet. 1994. Random Testing. In *Encyclopedia of Software Engineering*. Wiley, 970–978.

[36] Sean Heelan, Tom Melham, and Daniel Kroening. 2019. Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1689–1706. https://doi.org/10.1145/3319535.3354224

[37] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/3460319.3464795

[38] Peter Hlavaty. 2022. bananafzz. https://github.com/rezer0dai/bananafzz. [Online; accessed 10 April. 2022].

[39] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[40] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. 2020. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1613–1627. https://doi.org/10.1109/SP40000.2020.00063

[41] Tim Newsham Jesse Hertz. [n.d.]. TriforceLinuxSyscallFuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer. [Online; accessed 10 April. 2022].

[42] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[43] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[44] LLVM. [n.d.]. SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html. [Online; accessed 10 April. 2022].

[45] LLVM Project. [n.d.]. LibFuzzer - Value Profile. https://llvm.org/docs/LibFuzzer.html#value-profile. [Online; accessed 10 April. 2022].

[46] LLVM Project. [n.d.]. [libFuzzer] Port to Windows. https://reviews.llvm.org/D51022. [Online; accessed 10 April. 2022].

[47] LLVM Project. 2018. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. [Online; accessed 10 April. 2022].

[48] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[49] Dominik Maier, Otto Bittner, Marc Munier, and Julian Beier. 2022. FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols. In *Workshop on Binary Analysis Research (BAR), 2022*.

[50] Dominik Maier, Fabian Fäßler, and Jean-Pierre Seifert. 2021. Uncovering Smart Contract VM Bugs Via Differential Fuzzing. In *Reversing and Offensive-Oriented Trends Symposium* (Vienna, Austria) *(ROOTS'21)*. Association for Computing Machinery, New York, NY, USA, 11–22. https://doi.org/10.1145/3503921.3503923

[51] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/woot19/presentation/maier

[52] V. Manes, H. Han, C. Han, S.K. Cha, M. Egele, E. J. Schwartz, and M. Woo. 5555. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 01 (oct 5555). https://doi.org/10.1109/TSE.2019.2946563

[53] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with data dependency information. In *EuroS&P 2022, 7th IEEE European Symposium on Security and Privacy, 6-10 June 2022, Genoa, Italy*, IEEE (Ed.). Genoa.

[54] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA.

[55] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[56] Keean Schupke Oleg Kiselyov, Ralf Laemmel. 2004. HList: Heterogeneous lists. https://hackage.haskell.org/package/HList. [Online; accessed 10 April. 2022].

[57] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security*. Paper=https://download.vusec.net/papers/parmesan_sec20.pdfCode=https://github.com/vusec/parmesan

[58] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360600

[59] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941681

[60] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[61] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Network and Distributed System Security Symposium*. Network & Distributed System Security Symposium.

[62] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[63] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2795–2809. https://www.usenix.org/conference/usenixsecurity21/presentation/salls

[64] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Fuzzing on Interface Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[65] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 167–182.

To appear: CCS '22, November 7-11, 2022, Los Angeles, U.S.A.

Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti

[66] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing Vulnerabilities Statistically From One Exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 537–549. https://doi.org/10.1145/3433210.3437528

[67] Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. 1994. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, Tennessee, USA) *(SIGMETRICS '94)*. Association for Computing Machinery, New York, NY, USA, 171–180. https://doi.org/10.1145/183018.183038

[68] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective Grammar-Aware Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 244–256. https://doi.org/10.1145/3460319.3464814

[69] Robert Swiecki. [n.d.]. Honggfuzz. https://github.com/google/honggfuzz. [Online; accessed 10 April. 2022].

[70] Dmitry Vyukov. [n.d.]. syzkaller - kernel fuzzer. https://github.com/google/syzkaller [Online; accessed 10 April. 2022].

[71] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. 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)*. USENIX Association, Chaoyang District, Beijing, 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[72] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/reinforcement-learning-based-hierarchical-seed-scheduling-for-greybox-fuzzing/

[73] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.

[74] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[75] Michał Zalewski. 2014. Fuzzing random programs without execve(). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html. [Online; accessed 20 Dec. 2021].

[76] Michał Zalewski. 2016. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt. [Online; accessed 10 April. 2022].