École Polytechnique Fédérale de Lausanne

LangFuzzBench: Benchmarking Fuzzers for Structured Text Input Software

by Milan Duric

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Cono D'Elia Daniele
External Expert

Chibin Zhang
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 27, 2024

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Chibin Zhang and Prof. Dr. Mathias Payer, who have supported me and provided guidance during the course of this project.

Lausanne, June 27, 2024 Milan Duric

Abstract

Fuzzing has proven to be one of the most effective dynamic software testing techniques for bug finding. Yet, widely used target-agnostic fuzzers that mutate the input queue at byte-level struggle to find bugs in targets that accept highly-structured text input. To address this deficiency, the research community has worked on developing fuzzers that use a model of the language expected by the target in order to guide their mutation strategy. This model is often referred to as the grammar, hence these fuzzers are known as grammar-aware mutational fuzzers. However, many of these fuzzer prototypes have not been evaluated systematically enough. Firstly, they have all been evaluated on a disparate set of targets using different metrics, making it difficult to reason about their performance. Secondly, only a handful of these fuzzers have evolved beyond research prototypes and have been integrated to standardized benchmarks like Google's Fuzzbench. Specifying grammars is a time-consuming and error-prone process that requires repeated human effort. Moreover, current fuzzing benchmarks have slow build times and have not been optimized for incremental recompilation scenarios, which increases the duration of the feedback loop and impedes prototyping.

To tackle this problem, we adopt a new design for building fuzzing benchmarks and implement it in our prototype - LangFuzzBench. Our design is focused on breaking the monolithic benchmark build process into many smaller, loosely-coupled components that can in most cases be recompiled independently of one another. Most notably, we break the target build phase into two components - target compilation and target instrumentation. The former remains completely independent of the choice of the grammar-aware fuzzer. Thus, on a single system, LangFuzzBench really needs to compile each target only once, regardless of how many fuzzers we integrate it with. This leads to considerable time reductions when recompiling or building benchmarks incrementally. For example, our results show that it takes three times less on average to build a benchmark that integrates a new grammar-aware fuzzer with a target that was already fuzzed once on the system. This is not only enabled through clear separation of target compilation and instrumentation phases, but also through the use of the Nix package manager, that decouples our benchmark components by design.

The agility of the LangFuzzBench's build process allowed us to run many fuzz campaigns. Our results show that hybrid fuzzer setups that employ both byte-level and grammar-aware mutations outperform purely grammar-aware setups of the same fuzzers in both bug-centric and code-centric metrics. Furthermore, we observe considerable improvements for corpora-free fuzzers when enhancing grammars with language vocabulary. However, this strategy does not work for fuzzers that use a pre-collected corpus, as it needs to be adapted to remain parsable by the enhanced grammar.

Contents

Chapter 1

Introduction

The increasing complexity of modern software necessitates constant improvement of software testing techniques. New software gets constantly released, but critical security bugs remain in several years old code. Closing this gap has become a priority for both software engineers and security researchers in order to ensure reliable and secure software. Therefore, significant effort has been invested over the recent years in researching various automated software testing techniques. Out of all those techniques, fuzzing remains the most promising to this point.

Initially, one of the main design goals when building fuzzers was to build tools that are target-agnostic. Both AFL [1] and its successor AFL++ [12], which remain one of the most successful fuzzers to this date, were designed as universal fuzzers that can fuzz any target. However, years of research have shown that these fuzzers do not perform as well against targets that accept highly structured text input. These targets include any type of language processors, such as assemblers, compilers or interpreters of programming languages, document or data interchange format processors etc. Such targets are difficult to fuzz because they typically employ highly rigid lexical and syntactical checks against the passed input before triggering any real functionality (hence the name highly structured software). As most of the fuzzers are target-agnostic, they typically mutate the input queue on the level of bits or bytes. These low-level mutations of test cases typically break the syntactical validity of the input, and the fuzzer spends most of the time producing test cases that are rejected at the early stage by the lexer or parser of the target. Although the goal of fuzzing is to test program's behavior when faced with unexpected inputs, the fuzzer cannot achieve this if the actual code never gets executed. Furthermore, it is not possible to simply patch the target and remove any lexical and syntactical checks, like one would remove cryptographic checksum or magic byte sequence checks when fuzzing simpler targets. This is because the results of lexical and syntax analysis are actually consumed by the deeper code, so they cannot be omitted.

Over the past years, the security research community has recognized the need for new approaches to mutational fuzzing that will be suitable for highly structured text input targets. Many

new solutions have been implemented, however, only few have evolved beyond research prototypes and got widely accepted by the community. Only a handful of such fuzzers have been integrated to standardized benchmarks like Google's Fuzzbench [22]. The main reason for this is that these fuzzers are difficult to start working with, as they typically require *hand-crafted grammars, dictionaries or similar specifications of the target language* from the beginning. Defining these elements typically requires manual, human work and is very time-consuming. On top of that, many of these fuzzers have not been evaluated systematically enough. Firstly, they have all been run against a disparate set of targets. Secondly, even if some fuzzers have been run against the same set of targets, they have all been configured with different grammar specifications, making it very difficult to judge if better results arose due to innovative mutation strategies of the fuzzer concerned or simply due to investing more time in defining high-quality grammars. Finally, different performance metrics were used for evaluation. Even with standardized metrics, such as edge coverage, different implementations may yield different results. All these discrepancies make it very difficult to compare the performances of these fuzzers, both quantitatively and qualitatively.

To address these shortcomings, we developed LangFuzzBench - a fuzzing benchmark for evaluating grammar-aware fuzzers. To the best of our knowledge, this is the first benchmark specifically designed for evaluating grammar-aware fuzzers. LangFuzzbench offers an easy mechanism to integrate new grammar-aware fuzzers and targets, scripts to run the fuzzing campaigns, and a toolkit for post-campaign metrics collection and evaluation. The main goals of LangFuzzBench design was usability, extensibility and reproducibility. For packaging fuzzers and targets, LangFuzzBench relies on Nix [23] - a package manager for reproducible and reliable builds and deployments. Nix takes care of managing all the project dependencies and ensures all build artifacts are deterministic. It also offers many reusable build recipes that eliminate a lot of the manual grunt work required when building projects from source. This allows users to focus on real tasks, like developing new grammars and dictionaries, and not waste time on fixing broken benchmark builds. This is particularly useful when integrating research projects with rigid, unhardened build processes. Finally, the reproducibility of our benchmark builds enables seamless streamlining of the entire benchmarking process.

Another important aspect of LangFuzzBench is the speed and the agility of its build process. When starting a fuzzing campaign, one must build many components. These components, at the very least, include the fuzzer program, the target program and any other utility required by the used framework (for example, Magma's monitoring utility in Magma [17]). When prototyping fuzzing projects, it is common to make many small changes to individual components in order to see if they bring any tangible benefits. It is therefore imperative to ensure repeated and incremental builds are efficient. Long build processes increase the time of the feedback loop and discourage many users from making their own contribution. LangFuzzBench addresses this problem in two ways. Firstly, by using the Nix build system, we ensure different components of the benchmark are decoupled from one another by design. What this means is that, once a certain target and certain fuzzer have been compiled for the first time on a system, subsequent changes to one of those two components will not require recompilation of the other in order to start the fuzzing campaign on the same system. This

is in contrast to other fuzzing benchmarks like Magma and Fuzzbench, where all components of the campaign are tightly coupled by being compiled together as part of the build process of a single Docker container. In this situation, the time savings of recompilation are driven purely by Docker's caching mechanisms, which is purely determined by the linear order in which components are built in the Docker file. Secondly, we improve LangFuzzBench's build times by further decomposing standard components into subcomponents. For example, when preparing a target for a fuzzing campaign, it is necessary to *a) compile* and *b) instrument* the target. One needs to observe that those are two separate stages. The former stage (compilation), is completely independent of the choice of the fuzzer. The latter stage, however, depends on the choice of the fuzzer. If certain changes to the fuzzer are made, the fuzzer naturally has to be recompiled. The target, on the other hand, only needs to be reinstrumented. Therefore, by breaking the target build into two independent phases, the target compilation and target instrumentation, we further improve benchmark build times in cases of repeated and incremental changes of the fuzzer. This is achieved with the help of LLVM link-time-otpimization (LLVM LTO) [19], which allows us to compile the target only once as bitcode, and then instrument it any number of times we require. Further time savings are also achieved by design with the help of Nix, as already indicated. For example, the compiler needed for producing the target's bitcode is also built as a separate Nix derivation, completely independent of any other components of the benchmark.

These features of LangFuzzBench allowed us to integrate several grammar-aware fuzzers and run extensive fuzzing campaigns against a wide range of targets. The agility of the LangFuzzBench's build process allowed us also to try multiple grammar versions for the same fuzzer-target integrations. One of the main takeaways we would like to highlight is that grammar-aware mutational fuzzing is by no means a replacement for byte-level mutational fuzzing. Indeed, many grammar-aware fuzzers that employed exclusively grammar-abiding mutations performed no better in our benchmark than vanilla AFL++, even after evaluating both code-centric and bug-centric metrics. Although this can always be attributed to the quality of the grammar, other research also agrees that pure grammar-aware fuzzing suffers greatly from the bias factor. This not only prevents the fuzzer from exploring target's code related to lexing and parsing (hence lower code coverage), but also struggles to find bugs that simply cannot be triggered with grammar-abiding test cases (hence lower bug count). To prove this claim, we analyzed edge coverage and number of discovered bugs in campaigns that used the two fuzzers that can be configured to run either with or without byte-level mutations - Autotokens (Token-Level AFL) and Polyglot. Out of the 30 campaigns considered (15 per fuzzer), the setup that combined both byte-level and grammar-abiding mutations achieved greater edge coverage in 25 campaigns (83%) than the setup that performed purely grammar-abiding mutations. Furthermore, in both cases (for both fuzzers) we were able to find more bugs when combining byte-level and grammar-aware mutations. This proves that one cannot always completely attribute the lack of success to the quality of the grammar, and that mutation scheduling is just as equally important.

The second point we would like to highlight is the necessity of understanding the fuzzer before

building the grammar it should use. Many previous research tries to simplify this problem by drawing universal conclusions about how to approach defining grammars used for fuzzing. One such postulate is that the vocabulary used in the grammar is much more relevant than the comprehensiveness of the production rules being used. This is indeed true for corpora-free fuzzers such as Nautilus [7] and Grammar-Mutator [4] that use the grammar for test case generation. However, our results shows that fuzzer's that rely on a pre-collected corpus, like Polyglot [8], actually perform better (or are at least easier to make perform better) when used with comprehensive, highly generic grammars designed for parsing. This is due to the interplay between the grammar and the corpus - test cases need to be parsable by the grammar in order to be consumed. Extending the grammar carelessly with vocabulary may render many test-cases unparsable and unusable. However, semantically diverse corpus should play an even bigger role in grammar-aware fuzzing as an attempt to counter the negative bias introduced by humans when defining grammars. To prove this, we analyzed the effects of using a vocabulary-enhanced grammar on the edge-coverage results of Polyglot. We separated the targets into two groups of six. In the first group, we simply added vocabulary to the grammars without adapting the corpus. In the second group, we added vocabulary to the grammars but also ensured the corpus remained parsable. In total, targets from the first group experienced an average drop in delta edge coverage (total coverage discovered during the campaign, without the trivial coverage contributed by the corpus) of 0.85 percentage points, whereas targets from the second group experienced an average delta edge coverage increase of 1.29 percentage points. When examining total coverage, these percentages are even higher.

To summarize, the contributions of this paper are the following:

- We propose a new design for building fuzz benchmarks, focused on loose coupling of individual benchmark components that enables quicker and more reliable benchmark builds. We demonstrate it is superior to Docker-based approaches used by other fuzzing benchmarks in cases of repeated recompilation and constant addition of new fuzzers and targets to the benchmark.
- We implement this design in our prototype, LangFuzzBench, tailoring it specifically for grammar-aware mutational fuzzers. To the best of our knowledge, this is the first fuzzing benchmark focused exclusively on these types of fuzzers. We integrate 8 different grammar-aware mutational fuzzers and 15 different targets into LangFuzzBench. LangFuzzBench remains open-source and fully extensible.
- We conduct multiple fuzzing campaigns using LangFuzzBench, which leads us to the discovery of 32 new bugs: 21 of them in Micropython, 5 in Lua, 2 in the Solidity compiler, 2 in Hermes, 1 in MRuby and 1 in Yara.
- We test different approaches to defining grammars for grammar-aware fuzzing, evaluate them, and draw the conclusions that will facilitate the work with these fuzzers in the future, but also help the design of new tools for fuzzing targets that accept highly structured text input.

Chapter 2

Background

In this section we will introduce the concept of fuzzing. We will also explain why traditional byte-level mutation operators are inadequate for fuzzing complex language processors. Then, we will introduce the idea of grammar-aware fuzzers and illustrate how such fuzzers overcome the deficiencies of byte-level fuzzers. Finally, we will provide a brief overview of Nix [23] - a reproducible build system that represents the basic building block of LangFuzzBench.

2.1 Fuzzing

Fuzzing is an automated, dynamic software testing technique. The main reason behind its popularity nowadays is the fact that it fares very well with the increasing complexity of modern software. Modern software systems, like web browsers, have become notorious for their complexity. Their code base can contain tens of millions of lines of source code. This makes it very difficult for techniques like formal verification or symbolic execution to reason about the security of such systems comprehensively. Formal verification may be feasible only for minuscule and well-defined subcomponents, whereas symbolic execution would inevitably succumb to the path explosion problem. On the other hand, static analysis still has its place in testing such systems, but it needs to be complemented with a dynamic testing counterpart, like fuzzing, in order to account for dynamic behaviors that manifest only at runtime.

In fuzzing, the target program is continuously tested within a loop against random inputs. Modern fuzzing is a fully automated procedure, which means that the process of picking a random input, feeding it to the target program and noting down the results of execution is done by the fuzzer program, and not by a human operator. This is a very useful feature for testing highly-complex and large systems, as the fuzzer can be scheduled to run for days independently, without any human interaction.

The main intent behind picking a random input is to see how the target program behaves when facing anomalous data or unexpected conditions. Such conditions are usually the ones in which many programs exhibit buggy behavior. Modern fuzzers usually follow an evolutionary input generation strategy. This means that inputs that trigger crashes or otherwise interesting behavior are placed back into the fuzzing queue in order to be used again in the upcoming iterations. Before being passed again to the target program, the inputs will be evolved (or mutated) by the fuzzer in an attempt to discover new interesting behavior. The exact set of mutations that will be applied depends on the mutation strategy of the fuzzer, which becomes a very important decision when fuzzing programs that normally expect *highly structured inputs*.

2.1.1 Coverage-guided fuzzing

Triggering program crashes is not the only way to detect interesting input samples, as crashes are a very sparse signal on their own. Modern fuzzers also gather information on which parts of the program a particular input executed. If a particular input has discovered part of the code that was previously not executed during the same fuzzing campaign, the input is considered as interesting and placed back in the queue for subsequent mutations. In this manner, the fuzzer prioritizes inputs that trigger new behavior in the target program.

The information on which part of the target program was executed is usually referred to as program coverage. Depending on the level of granularity of such information, we differentiate several types of coverage. The type that provides the best cost-to-value trade-off is edge coverage. In edge coverage, locations of code that correspond to edges of the target program's control flow graph are instrumented at compile-time with code that will collect coverage information. Coverage-guided fuzzing was popularized by AFL [1], which demonstrated it can find real-world critical vulnerabilities effectively.

2.1.2 Byte-level mutational fuzzing

Testing a program continuously with the same set of inputs will not bring new program coverage. Input samples need to be mutated in order to discover new parts of the target's code base. Different fuzzers employ different mutation strategies, which we refer to as *the set of mutational operators* supported by the fuzzer.

Byte-level mutational fuzzing is a type of fuzzing in which the mutation operators of the fuzzer are applied on the level of bytes or bits of the input sample. Such fuzzer implement a relatively simple mutation phase that is supposed to be agnostic of the target being fuzzed. They perceive the input sample simply as a stream of bytes, without maintaining any notion about the structure of the inputs the target program is expecting. Typical mutation operators include byte or bit flips, byte insertions or removals, as well as arithmetic operations like addition or subtraction performed at

the level of bytes.

AFL++ [12] is a *byte-level mutational* fuzzer that groups its mutation operators into two different categories. The first category, also known as the *deterministic* stage, includes highly deterministic set of operators that have been empirically shown very effective. These operators do not change the size of a single input sample. They include deterministic bit and byte flips, additions, and substitution with integers from a set of interesting values (e.g. -1, 1, INT_MIN, INT_MAX, ...) etc. It was shown that these deterministic steps tend to generate compact test cases that ultimately lead to small structural differences between crashing and non-crashing inputs in the future, which makes it easier to reason about crashes after the campaign finishes. In the second category, also referred to as *havoc* phase, mutation operators are randomly stacked and applied in batch mode. This phase can alter the size of individual input samples, by removing or adding new bytes. Finally, AFL++ also may merge two existing inputs into a single input and then apply the havoc stage, which is also known as the *splicing* operator.

The main challenge with using *byte-level mutational* fuzzers is that the mutated inputs will be predominantly invalid. Depending on the target being fuzzed, this may result in very poor coverage, in case the target employs structural checks against the passed input and terminates before triggering any 'deeper' code. In some cases it is possible to overcome this problem by patching the target to eliminate such structural checks. However, this is only possible if such checks are simple and do not interact or depend on other parts of the code. Good examples are cryptographic checksums in targets that implement network protocols, or magic byte sequence checks in image parsing libraries. However, if the target truly expects highly structured inputs, like document format processors or scripting language interpreters, such ad-hoc solutions are not feasible. *Byte-level mutational* fuzzers are not suitable for such targets, as their mutation operators will break the expected format of the input and terminate all executions in the early processing stages of the targets, such as lexical or syntax analysis at best.

2.1.3 Grammar-aware mutational fuzzing

Unlike *byte-level mutational* fuzzers, *grammar-aware mutational* fuzzers maintain a structural model (a grammar) of the input expected by the target program. This model is used to guide the mutation operators when evolving the input queue of the fuzzer. Provided that the grammar given to the fuzzer accurately reflects the grammar used in the target's implementation, *grammar-aware mutational* fuzzers will generate only syntactically correct inputs. This makes *Grammar-aware mutational* fuzzers, at least conceptually, a great choice when fuzzing programs that expect highly structured input, as their mutation operators will not break the syntactical correctness of the input and will allow the fuzzer to explore deeper and more interesting parts of the target that go beyond lexical and syntax analysis.

Nevertheless, these fuzzers face their own set of challenges. In order to fuzz a target with a

grammar-aware mutational fuzzer, it is typically required to have a prepared grammar even before the campaign can start. Therefore, fuzzing exotic targets that do not publish the grammar implemented by their parser becomes infeasible from the outset. Furthermore, different *grammar-aware mutational* fuzzers expect grammars to be specified in different formats. For example, Polyglot [8] and Superion [32] can work only with grammars specified in the ANTLR4 [5] format. Conversely, Nautilus [7] uses a customized Python syntax to specify the grammar. Likewise, Grammar-Mutator [4] uses a custom JSON syntax for defining grammars, whereas Gramatron [31] uses grammar automatons. Furthermore, different *grammar-aware mutational* fuzzers impose different restrictions regarding the expressiveness of the grammars. For example, Nautilus allows the use of regular expressions when specifying terminal symbols, whereas Grammar-Mutator does not. They also enforce different naming conventions for non-terminal rule definitions. All these details quickly become a burden because grammar development requires human effort, which can be expensive and error-prone. However, the success of a grammar fuzzing campaign depends heavily on the quality of the provided grammar.

Another major limitation of *grammar-aware mutational* fuzzers is the introduction of the bias factor. After starting a fuzzing campaign, the *grammar-aware mutational* fuzzer will only generate inputs that completely adhere to the specified grammar. This means that the *grammar-aware mutational* fuzzer will not be able to discover bugs that cannot be expressed by the specified grammar. However, bugs in lexers and parsers of language processors are still a common sight (CVE-2018-8279, see Listing 1.). This problem becomes even more amplified if the specified grammar also ensures semantic correctness (in addition to syntactical correctness) of the generated inputs. In such case, bugs that cannot be reached by semantically correct input will be overlooked by the fuzzer. Unsurprisingly, many serious security vulnerabilities in language processors actually come from improper handling of semantically invalid input (CVE-2017-8645, see Listing 1). For this reason, *grammar-aware mutational* fuzzers usually need to relax the grammar constraints during input generation. Alternatively, the users of such fuzzers need to beware of the bias factor and design more relaxed grammars. Finally, a viable alternative is to integrate the *grammar-aware mutational* fuzzer as an additional mutation phase of an existing *byte-level mutational* fuzzer. What this means is that the fuzzer will employ both byte-level and grammar-aware mutation operators in a round-robin fashion. Indeed, many of the fuzzers included in our benchmark already have integrations developed for the AFL++ custom mutator API [3].

2.2 Nix package manager

Nix is a powerful package manager built with the aim of flexibility, reproducibility and ease of use. It adopts a fully functional programming language model for defining and building packages. Nix packages are built in sandboxed environments, isolated from one another and from the host system. Traditional package managers often install packages and their dependencies in shared

Listing 1 Proof of concept code for CVE-2018-8279 (left) and CVE-2017-8645 (right). Both CVEs affected different versions of ChakraCore. The left example contains a syntax error, as the built-in await token should not be allowed in the given context. However, the vulnerable ChakraCore implementation proceeded to generate bytecode for the faulty JavaScript, ultimately leading to type confusion and remote-code execution. Similarly, the right example contains a type error - the class constructor cannot be invoked without the 'new' keyword. This bug could also lead to arbitrary remote-code execution. The goal of the security research community is to detect such bugs with automated fuzzing. However, a grammar that generates only syntactically and semantically correct inputs cannot guide the fuzzer to such faulty code paths.

```
async function trigger(a = class b {
    [await 1]() {
    }
}) {
}

let spray = [];
for (let i = 0; i < 100000; i++) {
    spray.push(
        parseFloat.bind(
            1,
            0x1234,
            0x1234,
            0x1234,
            0x1234
        )
    );
}

trigger();
```

```
class MyClass {
    f(a) {
        print(a);
    }

    constructor() {
        'use asm';
        function f(v) {
            v = v | 0;
            return v | 0;
        }
        return f;
    }

    f2(a) {
        print(a);
    }
}

MyClass(1);
```

system directories. This quickly leads to the "dependency hell" problem, which happens when different packages require different versions of the same dependency in order to build properly. In Nix, packages are built in a directory determined by the cryptographic hash of the package's entire dependency graph. This allows users to seamlessly install multiple different versions of the same package or dependency via Nix, which automatically takes care of resolving the system location. Moreover, it also helps in making builds *fully reproducible*. These features come very handy when building a system like a fuzzing benchmark.

Another benefit of Nix is that it eliminates a lot of the boilerplate code and grunt work needed to build a complex package from source. This is possible because many projects follow a well-defined sequence of steps when being built. For example, a typical autotools-based project would start by preparing the build environment, move on to the actual compilation of the source code, optionally run some tests afterwards, and finish by copying the resulting binaries to the specified system directory. Nix exploits these repeating patterns by defining them as reusable functional recipes. These recipes, also known as Nix derivations, are reusable Nix functions that execute a predefined sequence of steps, yet are highly customizable (see Listing 2). After executed, results of individual derivations get cached and can be later re-used even in different package builds and nix profiles. As will become clear in the remaining sections of this paper, this is a major performance advantage of Nix over some other systems, like Docker, which have admittedly not been designed as package managers, yet are often being used in systems-oriented projects as a way to achieve isolation and reproducibility.

Listing 2 Example Nix script for building CPython from source. At the top we see a set of packages that Nix will automatically pull from the Nix Package Repository [24]. These packages are then declaratively specified as built-time dependencies of our package. Nix expression *stdenv.mkDerivation* contains a recipe for building autotools-based projects. This recipe automatically invokes 7 different phases when building our package. One of these phases, the *patchPhase*, which is used to change some aspects of the source code after downloading it but before starting the compilation, is overridden in our configuration.

```
{ stdenv, fetchFromGitHub, pkg-config, python3, zlib }:

stdenv.mkDerivation rec {
  pname = "python3.6";
  version = "v3.12.2";

  src = fetchFromGitHub {
    owner = "python";
    repo = "cpython";
    rev = version;
    hash = "sha256-hhLY1nfCMaGuj2N2j/ag9yEyaksIiAtO3rgHfrhpYZo=";
  };

  enableParallelBuilding = true;
  nativeBuildInputs = [ pkg-config python3 zlib.dev ];

  prePatch = ''
    sed -i "s/perf_trampoline=yes/perf_trampoline=no/g" configure
  '';

  configureFlags = [
    "--disable-shared"
    "--disable-test-modules"
    "--with-static-libpython=yes"
    "--with-ensurepip=no"
    "ax_cv_c_float_words_bigendian=no"
  ];
}
```

Chapter 3

Design

In this section we provide more details on the design of LangFuzzBench. We provide an overview of the benchmark, focusing on its goals and constraints. We then discuss the integration aspects of different fuzzers, particularly the design of the grammars.

Before proceeding, we introduce the following terminology that will be used in the reminder of this work:

- ***Target***: The program being fuzzed.
- ***Fuzzer***: Component that provides either the grammar-aware custom mutator or the full fuzzing driver (e.g. afl-fuzz).
- ***Toolchain***: If the *fuzzer* provides just the custom mutator, then the *toolchain* includes both the instrumenting compiler and the fuzzing driver. Otherwise, if the *fuzzer* provides the full fuzzing driver, then the *toolchain* includes just the instrumenting compiler (e.g. afl-cc).

3.1 Goals

Our goal is to provide a framework that will allow security researchers to streamline the evaluation of different grammar-aware fuzzers. The framework should allow users to seamlessly add new fuzzers and targets, provide scripts to build the benchmarks and run the fuzzing campaigns, as well as a toolkit for post-campaign metrics collection and evaluation. Thus, the main goal of the framework is **easy extensibility**. Fuzzing projects, as well as some targets like JavaScript engines, can be quite complex to build from source. They may require a lot of different dependencies that may conflict with existing packages available on the user's host system. Other fuzzing frameworks typically overcome this issue by packaging benchmarks as isolated Docker containers, thus paying the expensive price

of tight coupling of benchmark components in exchange for relatively reliable and robust builds. Our goal is to avoid paying this price and maintain the loose coupling of benchmark components while also improving the reliability of the benchmark builds.

Loose coupling of benchmark components is a necessary design decision in order to achieve our second goal, which is **improved build speeds in case of recompilation of particular components**. We expect users to not only add new fuzzers and targets, but to also frequently make changes to them (e.g. using a different sanitizer, changing the grammar of the fuzzer etc.). In such cases, we do not wish to pay the full price of rebuilding the entire benchmark from scratch. However, certain dependencies between components are inevitable. For example, when adding a new target or modifying its grammar, certain fuzzers need to be recompiled from scratch because they need full grammar specification at compile-time. Likewise, a new fuzzer may require custom environment variables to be set when instrumenting targets, making the previous target compilations useless. To tackle these dependencies, we adopt two approaches.

Firstly, we distinguish situations in which the compile-time dependency is based only on the **name** of the component as opposed to its actual content. Taking the previous example, the compilation of the target is dependent only on the name of the fuzzer, because from this name we can accurately derive the values of the custom environment variables needed for instrumentation. LangFuzzBench ensures the expensive target compilation is not triggered on repeated fuzzer recompilations due to its configuration changes. It is triggered only when building the fuzzer with the particular name for the first time on the system.

Secondly, to further improve benchmark build speeds in cases of recompilation, we further decompose benchmark components to lightly-coupled subcomponents. The most notable insight we make is that the target compilation and instrumentation are two completely different and independent phases. Indeed, the compilation of the target should really have nothing to do with the fuzzer, whereas the instrumentation depends only on the name of the chosen fuzzer. Breaking the target into these two components allows us to compile the target only once, while instrumenting it for different fuzzers arbitrary number of times. More details on the exact components of our benchmark will be presented in the next section.

3.2 System overview

Each fuzzer-target integration in LangFuzzBench needs to go through three separate, logical phases in order to be built. Note that these phases are not explicitly manifested in implementation but are still very useful to conceptualize. These phases are depicted on Figure 3.1 as colored boxes. Each phase consists of one or multiple components. These components are reusable units of LangFuzzbench that accept certain arguments and produce some result. They are shown on Figure 3.1 as numbered, white rectangles, whereas argument passing is represented with directed lines.

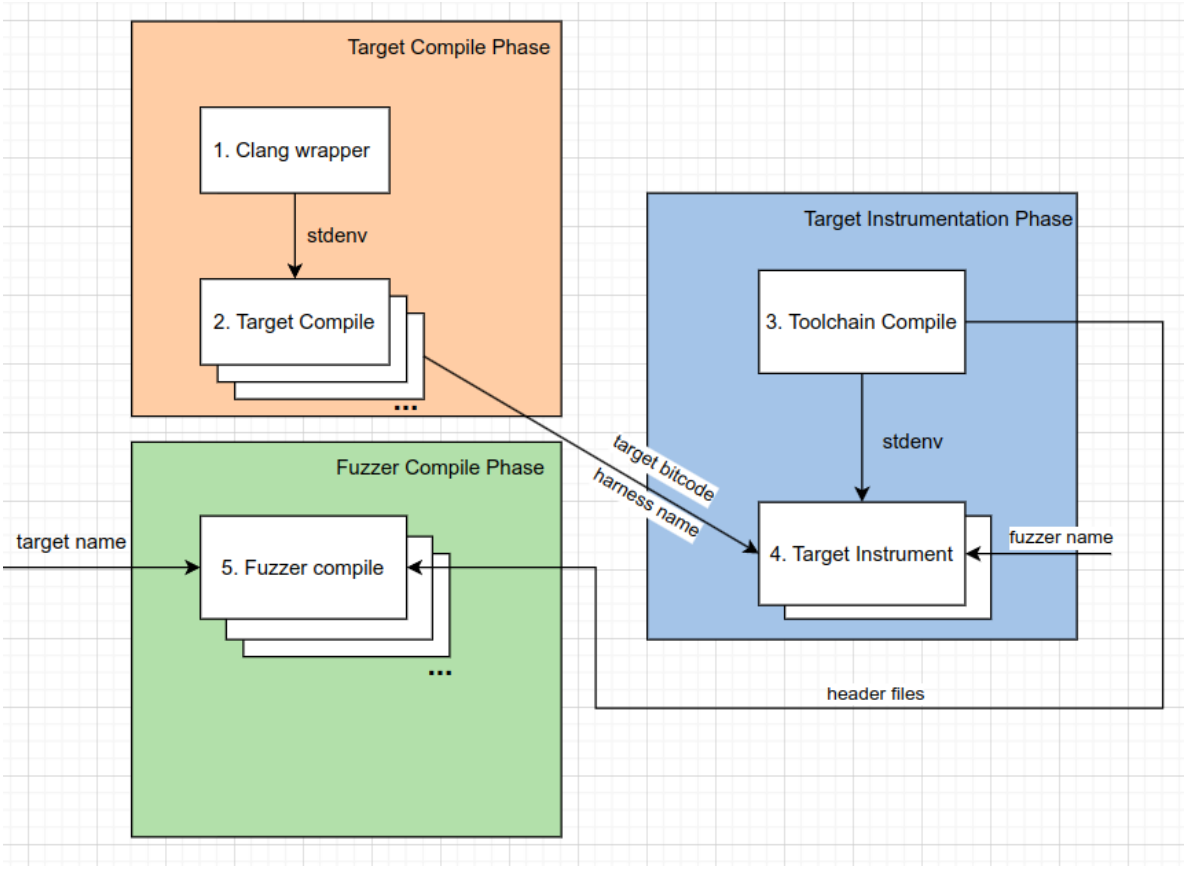

Figure 3.1 – Overview of the benchmark build process

Finally, each of these units may actually require one or variable number of actual Nix derivations to be produced. This aspect is depicted as stacked rectangles on Figure 3.1. If variable number of derivations is possible, then triple dots (...) are used. We now describe each conceptual phase and its units.

3.2.1 Target Build Phase

The aim of this phase is to produce reusable target bitcode using the LLVM link-time-optimization (LLVM LTO, [19]) that can later be instrumented by the toolchain in different ways depending on the choice of the fuzzer. The original design philosophy behind LLVM LTO was to enable powerful inter-modular optimizations to happen at link-time. This is useful because the linker observes multiple translation units and can hence perform more informed optimizations, as opposed to the compiler who translates and optimizes a single source file at a time. To achieve this, the output of the compilation phase (and the input to the linking phase) is now a specific intermediate representation (IR) of the source unit that we refer to as LLVM bitcode.

This phase consists of two separate units (see orange box on Figure 3.1). The first unit produces a wrapper script around the Clang compiler that will ensure necessary compiler and linker flags (CFLAGS and LDFLAGS) for using the LLVM link-time-optimization are passed when actually invoking the compiler. This script is then passed to the second unit that actually compiles the target and produces the LLVM bitcode. Note that the latter unit can produce different number of actual Nix derivations depending on the target being compiled, as indicated with stacked rectangles and triple dots on Figure 3.1.

The main benefit of this approach is that the target build phase remains completely independent of the fuzzer build phase. This can be observed on Figure 3.1 - indeed, there are no argument edges coming from the fuzzer build phase to the target build phase. Therefore, assuming we do not make any changes to the source code of the target or its compilation aspects, each target will really be compiled only once on our system, regardless of how many fuzzers we use. This feature becomes really useful for complex targets with hundreds of thousands of lines of code that are time-consuming to compile, like JavaScript engines. Note that we could have further optimized our design by completely omitting target compilation and simply delivering ready-made LLVM bitcode to the instrumentation phase. The reason why we chose not to do this is that many of these targets have highly customizable compilation options. These options allow users to include or completely omit certain plugins from the compilation. The number of potential LLVM bitcodes for a single target to keep track of would grow exponentially with the number of these options. Whether this feature would be useful depends on the nature of the use of the benchmark.

3.2.2 Target Instrumentation Phase

With the LLVM bitcode produced, we can now proceed with the target instrumentation phase. This phase consists of two units. The first unit compiles what we refer to as the toolchain. The toolchain, at the very least, produces the chosen instrumenting compiler needed to instrument the target. It can also produce additional binaries, such as the main fuzzing driver, that may be used for the campaign depending on the chosen fuzzer. Finally, some fuzzer builds also use the header files which were downloaded during toolchain compilation.

LangFuzzBench currently integrates only one toolchain, which is the AFL++ toolchain. This means that the main derivatives of this build unit are afl-cc and afl-fuzz. The choice of the toolchain to be used is dictated by the choice of the fuzzer. Currently we are unaware of any grammar-aware fuzzer that cannot work with the AFL++ toolchain. However, if any of them arise in the future, they can also be integrated to LangFuzzBench, because LangFuzzBench supports adding new toolchains. The choice of which toolchain to use for a particular benchmark in LangFuzzBench is made through a table lookup indexed by the name of the grammar-aware fuzzer.

The second unit in this phase is responsible for instrumenting and linking the LLVM bitcode using the instrumenting compiler built in the previous unit. This unit produces two Nix derivations. The first derivation creates a wrapper around a shell script that will actually invoke the instrumenting compiler, whereas the second derivation actually calls this wrapper script. Three things need to be done in the wrapper script. Firstly, we need to manually set the LDFLAGS for the linker. This is achieved by iteratively reading shared object dependencies of the target using *ldd*. Secondly, we need to extract the LLVM bitcode into a separate file. LLVM bitcode is typically stored in a separate object file section called *.llvmbc*. We simply extract this section using *objdump*. Finally, we need to invoke the instrumenting compiler using the appropriate set of environment variables which are determined by the choice of the grammar-aware fuzzer. For this reason, the name of the fuzzer is shown as an argument to this unit on Figure 3.1.

3.2.3 Fuzzer Build Phase

The final phase of building the benchmark is responsible for building the grammar-aware fuzzer. These fuzzers typically require custom grammars or dictionaries in order to reason about the structure of the language expected by the target. However, some of these fuzzers require this information purely at run-time, whereas some also need it at compile-time. For example, Nautilus needs the grammar only at run-time, Autotokens needs the dictionary also only at run-time, whereas Polyglot needs the ANTLR grammar at compile-time in order to generate the lexer and the parser for processing the corpus. In order to simplify the benchmark and the process of adding new fuzzers, we designed LangFuzzBench by considering the worst-case scenario - that the grammar, dictionary or any other model of the structure accepted by the target is required by all fuzzers at compile-time.

More precisely, this dependency is resolved in a uniform manner by simply passing the name of the target being fuzzed to the Nix expression building the fuzzer. The fuzzer then proceeds to fetch the grammar, dictionary or anything else it needs from the expected location on the file system for that particular target. In this manner, we ensure that the fuzzer remains decoupled as much as possible from the target. For example, if we modify the target without touching its grammar (e.g. we enable thread safety for PHP by passing *–enable-zts* to the configure script), the fuzzer will not need to be recompiled. Similarly to the unit responsible for compiling the target and producing the LLVM bitcode, compiling the fuzzer can also result in varying number of Nix derivations produced.

3.3 Fuzzer integration

In addition do developing a Nix script for building the fuzzer, the process of integrating the fuzzer to LangFuzzBench also includes developing grammars, dictionaries or any other accompanying material required by the fuzzer that models the structure of the language expected by the targets. The success of the fuzzing campaign is highly dependent on the quality of this material. For this reason, we invested significant time in designing grammars and experimenting on this matter. However, the comparisons between fuzzers should also be drawn between the quality of their mutation operators, and not just the quality of the grammars they were fed with.

The grammars that we use as baselines are the official ANTLR grammars implemented by the parsers of the targets. These grammars were either collected from the target project's home page or from the following open repository [6]. The first problem when using these grammars is that they need be adjusted to the format expected by the fuzzer. Nautilus expects grammars in a custom Python format, whereas Grammar-Mutator expects the grammars in JSON. Certain ANTLR features simply cannot be expressed in these formats. For example, ANTLR grammars commonly employ custom C/C++ code in token or rule definitions in order to overcome certain constraints of the input language that simply cannot be expressed with a context-free grammar. A great example to think of is indentation tracking in Python. The number of (de)indentations the next line should have is highly dependent on the context - the line's location in the parent block. The full mechanism for handling this is specific to ANTLR and cannot be simply converted to other formats. On the other hand, some features are only supported by a subset of fuzzers from LangFuzzBench. For example, regular expressions for token definitions are allowed by Nautilus but not allowed by Grammar-Mutator. When converting baseline grammars, we aim to mimic the original ANTLR grammar as much as possible. Nevertheless, one should remain aware of the aforementioned limitations.

In addition to adjusting the baseline ANTLR grammars for each fuzzer, we also try to find ways to improve them so that they lead to more edge coverage or target crashes. Our hypothesis was that certain grammar design patterns work better for certain types of fuzzers. In the following subsections, we provide a taxonomy of the fuzzers in LangFuzzBench and explain how this impacts the grammars these fuzzers need to use.

3.3.1 Corpora-free generators

In this group we include fuzzers that do not require an ahead-of-time collected corpus in order to start the fuzzing campaign. In fact, these fuzzers use the provided grammar in order to generate the initial set of test cases that will serve as seed files. Nautilus and Grammar-Mutator fall into this group. Both fuzzers allow the users to customize the size of the initial corpus, but any other aspects of the desired corpus need to be specified indirectly through the provided grammar.

The first problem with using baseline grammars for this group of fuzzers is that they have been designed for **parsing** and not for **generating**. Therefore, these grammars do not include the necessary whitespaces in rule definitions. This is because the parsers that use these grammars operate on abstract syntax trees. However, grammar-aware fuzzers from this group operate on concrete syntax trees (also known as parse trees). With parse trees, the entire tree is usually directly "unparsed" to a string and fed to the target. Therefore, all necessary whitespaces need to be modelled explicitly through the grammar used for input generation in order to be present in the parse tree before being converted to the final string fed to the target. Listing 3 illustrates the impact on generated test cases when omitting and including whitespaces into Nautilus grammar definitions.

The second problem with using baseline grammars with these types of fuzzers is that such grammars typically lack the vocabulary of the language. By vocabulary we do not only mean reserved keywords normally present in the lexer file, but also names of built-in functions, built-in modules, globally accessible variables etc. These values are orthogonal to all the non-terminal production rules required for a comprehensive grammar. While the non-terminal production rules allow the fuzzer to explore the target's code base in **width**, built-in symbols allow the fuzzer to explore the target's code base in **depth**. Built-in symbols are particularly important for these kinds of fuzzers because they cannot rely on a real corpus. In fuzzing, the corpus typically provides a set of useful seeds that already at the beginning execute some deeper code, or at least have the potential of being mutated to such test cases. Without the corpus and with a highly generic grammar, it becomes very difficult for the fuzzer to produce semantically interesting test cases. Listing 4 illustrates the problem.

3.3.2 Corpora-dependent fuzzers

The second group of fuzzers includes fuzzers that do not use the provided grammar for generating test cases. Conversely, these fuzzers rely on an ahead-of-time collected corpus that needs to be fed to the fuzzer at the beginning of the campaign. The grammar is used purely to guide the mutation operators during fuzzing in order to ensure syntactical and, in certain cases, semantical correctness. Fuzzers integrated to LangFuzzBench that fall within this group are Polyglot and Superion.

These fuzzers do not suffer from the problem of explicit whitespace definitions. The grammars that these fuzzers rely on are used to parse the corpus and convert it to a custom intermediate

Listing 3 On the left we see an extract from the baseline version of the solidity grammar adapted for Nautilus. The *STRUCTDEFINITION* rule does not specify a space between the 'struct' token and the ID rule. If we look at one test case generated from this grammar, we see a bogus string ***struct****AJs2e4*. Parsing of such a test case will be immediately terminated by the parser, and any potentially interesting code that occurs afterwards in the same file will be useless. Orthogonal to this problem is the issue of redundant whitespaces, illustrated on the right side of the listing. In this example, the *ID* non-terminal is supposed to start with one of the characters from *ID1*, followed by a sequence of zero or many characters from *ID2*. However, we accidentally add a space in the definition of *ID* between *ID1* and *AUX_107*. Thus, the generated import statement in the test case is unparsable. The main takeaway from these two examples is that the problem of whether or not to include a whitespace when converting a grammar operating on abstract syntax trees to a grammar operating on parse trees cannot be completely automated. Thus, manual human inspection is required.

```
> cat solidity_grammar.py
...
ctx.rule(u'STRUCTDEFINITION',
    u'struct{ID}{L_BR}{AUX_18}{R_BR}')
ctx.rule(u'AUX_18', u'')
ctx.rule(u'AUX_18', u'{GROUP_21}')
...

> ./nautilus_gen solidity_grammar.py
structAJs2e4{}
import { calldata as Ir } from "9H3A";
...
```

```
> cat solidity_grammar.py
...
ctx.rule(u'IMPORT_STMT',
   u'import {STRING} as {ID};')
ctx.rule(u'ID', u'{ID1} {AUX_107}')
ctx.rule(u'AUX_107', u'')
ctx.rule(u'AUX_107', u'{ID2}{AUX_107}')
ctx.regex(u'ID1', '[a-zA-Z$_]')
ctx.regex(u'ID2', '[a-zA-Z0-9$_]')
...

> ./nautilus_gen solidity_grammar.py
import "2GtC" as Y si3F$;
enum e1 {};
...
```

Listing 4 Extract from the grammar for fuzzing SQLite adopted for Nautilus. The example illustrates how built-in function names can be included to the grammar, while still ensuring test cases with completely random function names get generated as well. We can see that the generated example contains two invocations of existing functions, *max* and *concat*, and one invocation of a completely random function *aWD1*. The latter invocation was generated by the *ANY_NAME* rule. Note that the signatures of the two existing functions are not respected in the generated example. Defining such a grammar would be too time-consuming and would arguably introduce too much negative bias that would be detrimental for the versatility of our fuzzer.

```
> cat sqlite_grammar.py
...
ctx.rule(u'FUNCTION_NAME', u'{ABS}')
ctx.rule(u'FUNCTION_NAME', u'{CONCAT}')
ctx.rule(u'FUNCTION_NAME', u'{MAX}')
...
ctx.rule(u'FUNCTION_NAME', u'{ANY_NAME}')
ctx.rule(u'ABS', u'abs')
ctx.rule(u'CONCAT', u'concat')
ctx.rule(u'MAX', u'max')
...
ctx.rule(u'ANY_NAME', u'{IDENTIFIER}')
...

> ./nautilus_gen sqlite_grammar.py
SELECT c2, c6, c3 FROM t1 WHERE c1 = (SELECT MAX(c2, 'as2fEe') FROM t2);
ROLLBACK;
SELECT aWD1(c6, c1), CONCAT(1) FROM t4;
COMMIT;
...
```

representation (IR) tree. When "deparsing" this tree into a string meant to be fed to the target, the fuzzers automatically add whitespaces between symbols.

However, the fact that the grammar is used for parsing the corpus represents a new challenge. Firstly, if a grammar rule is not triggered by at least one test case from the corpus, it will be completely ignored by the set of mutation operators during the fuzzing campaign. Secondly, if a certain test case cannot be parsed by the grammar, either because of a lexical or syntactical error, it is discarded. This requires users to adopt an additional processing step on the corpus in order to ensure such test cases do not occur, as they are useless. This is contrary to traditional *byte-level mutational* fuzzers like AFL++, where corpus preparation typically includes only a few minimization techniques. From the user's perspective, this can quickly get time-consuming and complex. For example, we have already mentioned that a typical way of improving the grammar is to enhance it with vocabulary. Vocabulary includes things like built-in function and module names, globally accessible variables etc. However, as illustrated in Listing 5, adding vocabulary implies changing the grammar in a way that may result in parts of the corpus being unparsable.

Listing 5 Extract from the grammar for fuzzing CPython adopted for Polyglot. The example illustrates difficulties that can arise when enriching a grammar originally designed for parsing with vocabulary. This problem arises because many languages allow users to override the meaning of built-in names in particular scopes. In the example test case, we define a custom function with the same name as a global built-in function. The provided grammar cannot parse such a test case. There are two possible solution to this problem. The first one is to ensure our corpus is free from such test cases. This would require manual patching of all unparsable test cases, which could be very time-consuming. The second solution is to always add vocabulary to the "most generic" non-terminal rule possible (in this case rule "name"). However, this becomes difficult to track with big grammars.

```
> cat cpython_parser.g4
...
atom
    : NUMBER
    | STRING
    ...
    | function_name LP argslist RP
    ...
    ;
function_name
    : ABS // added vocabulary
    | MAX // added vocabulary
    ...
    | name // universal grammar for parsing would only use this as function_name
    ;
// can we now parse a function definition with name 'max' or 'abs' ?
funcdef: DEF name parameters ('->' test)? ':' block;
name: NAME;
...

> cat cpython_lexer.g4
...
DEF: 'def';
ABS: 'abs';
MAX: 'max';
NAME: [a-zA-Z_][a-zA-Z0-9_]+ // simplified...
...

> cat ./corpus/sample1.py
def max(a,b): # unparsable funcdef, lexer will return token MAX instead of NAME
    if a >= b:
        return a
    else:
        return b
...
```

Chapter 4

Implementation

In this section we provide more details into the implementation aspects of building benchmarks with LangFuzzBench.

4.1 Target build phase

Despite build reproducibility, one of the often overlooked benefits of using Nix as the build system for larger projects is its succinctness. Nix offers a declarative and functional model for building packages. This model eliminates a lot of the boilerplate code and commands that need to be executed when building projects from source. This enables seamless automation of the build process and allows developers to focus on the actual problems, and not waste time troubleshooting broken builds.

The sequence of listings below show the crux of the target build phase. Listing 6 shows the code needed to produce the wrapper around the Clang compiler that contains the compiler and linker flags necessary to compile the target with LLVM LTO. The result of this derivation is basically a modified version of the publicly available *llvmPackages_16* Nix package that can be easily reused in the calling code. The calling code is shown in Listing 7. The first *callPackage* invocation builds the derivation from Listing 6. The *callPackage design pattern* [25] eliminates the tedious work of explicit argument specification - the arguments corresponding to the three parameters from Listing 6 are automatically deduced (and if necessary downloaded) by Nix based on their name.

The second *callPackage* invocation from Listing 7 builds the LLVM bitcode of the target. An example script called at this point is given in Listing 2. This script needs to be written by the user when adding a new target (in this case CPython) to LangFuzzBench. When doing this, the user does not need to think about the complexities of the clang wrapper or LLVM LTO. These aspects are completely hidden from the user through the *stdenv* Nix object, which contains references to

core dependencies (compiler toolchain, GNU make etc.) needed to build software. On top of this, Nix provides users with *recipes* (stdenv.mkDerivation) that automatically try to execute a set of well-known, sequential steps. These steps are pluggable and customizable by the user, as shown with the *prePatch* phase in Listing 2. These recipes are another mechanism that Nix uses to reduce the grunt work typically present when building complex projects from source. Finally, project build-time dependencies can be declaratively specified through the *nativeBuildInputs* option of the stdenv.mkDerivation recipe. Nix packages and offers more than 100 thousand dependencies through its centralized repository [24].

Listing 6 Clang wrapper: Nix script that wraps the clang compiler under a script that sets the necessary compiler and linker flags for LLVM LTO

```
{ llvmPackages_16
, wrapClang
, overrideCC
}:

let
cc = wrapClang {
    inherit llvmPackages_16;
    cflags = [ "-flto" ];
    ldflags = [ "--plugin-opt=-lto-embed-bitcode=optimized" ];
};
in
{
stdenv = overrideCC llvmPackages_16.stdenv cc;
}
```

4.2 Fuzzer build phase

When extending LangFuzzBench with a new grammar-aware fuzzer, the user first needs to write a script similar to the example we used for building the CPython target (Listing 2). From LangFuzzBench's perspective, building the fuzzer is by no means different from building a target - it is just a custom Nix script that uses the *stdenx.mkDerivation* recipe in order to produce some derivations. Secondly, the user needs to convert the grammars and dictionaries of all targets to the format expected by the new fuzzer. Finally, certain fuzzers may require custom options to be passed to the toolchain's compiler when instrumenting targets. If this is the case, the user needs to modify the main instrumentation script by adding a simple *if/else* block that will handle this case, as shown in Listing 8.

Listing 7 Extract from the main Nix script that builds the clang wrapper (ltoDer) and invokes the compilation of the target

```
...

# first build the clang version with the LTO plugin
ltoDer = callPackage ./core/lto_embed_bitcode { };

# build the target with .llvmbc section containing the bitcode
targetBitcodeDer = callPackage ./targets/${target} {
    inherit (ltoDer) stdenv;
};

...
```

Listing 8 Extract from the main instrumentation script that invokes the toolchain's compiler in order to instrument the target. The logic for setting *LDFLAGS* was omitted for brevity. In order to enable extending LangFuzzBench with new toolchains, we hide the actual compiler used through the *compiler* variable. However, we do assume that this compiler is a derivation of either clang or gcc. CMPLOG instrumentation, which requires the compilation of an additional binary, is also supported.

```
...
# set environment variables
if [[ "$fuzzer_name" = 'nautilus' || "$fuzzer_name" = 'superion'
    || "$fuzzer_name" = 'afl_compiler_fuzzer' ]];
then
    export AFL_LLVM_INSTRUMENT=afl
fi;

# invoke the instrumenting compiler. Use ASAN if it does not crash the fuzzer
if [[ "$fuzzer_name" = 'afl_compiler_fuzzer' ]];
then
    # these fuzzers had problems when targets were compiled with ASAN
    "$compiler" -g -o "$name" "$name".bc $ldflags
else
    "$compiler" -g -o "$name" -fsanitize=address "$name".bc $ldflags
fi;
cp "$name" "$1/bin/$name"

# Handle CMPLOG instrumentation
if [[ "$fuzzer_name" = 'autotokens_cmplog' ]];
then
    export AFL_LLVM_CMPLOG=1
    "$compiler" -g -o "$name"_cmplog -fsanitize=address "$name".bc $ldflags
    cp "$name"_cmplog "$1/bin/$name"_cmplog
    unset AFL_LLVM_CMPLOG
fi;

...
```

Chapter 5

Evaluation

In this section, we will validate the claims made in previous chapters. To formalize our evaluation methods, we define them through three research questions:

RQ1: *Does LangFuzzBench offer significant savings in time needed to rebuild the benchmark in cases of repeated additions or modifications of benchmark components? Is LangFuzzBench competitive to other fuzzing frameworks in this regard? Does it outperform them?*

RQ2: *Do grammar-aware fuzzers consistently outperform traditional, byte-level mutational fuzzers on a wide-range of targets that accept highly structured-text inputs?*

RQ3: *How do different grammar design strategies influence the performance of different types of grammar-aware fuzzers?*

5.1 RQ1: Benchmark build efficiency

To answer question **RQ1**, we measure the time and the number of Nix derivations that need to be rebuilt when a single component of a LangFuzzBench benchmark is modified and compare these measurements to the baseline, which is compiling the same benchmark from scratch on a new system with a completely empty Nix store. We analyze the two most relevant components of LangFuzzBench which we assume will be modified the most by users: the fuzzer and the target. We conduct these performance experiments on an ordinary personal laptop running Ubuntu Linux with 8 CPU cores and 16GB of RAM. We anticipate many users will first verify on their personal computers that their benchmarks can compile successfully, and only use dedicated hardware for actually running the campaigns.

5.1.1 Fuzzer build analysis

We consider two situations in this case. The first situation is adding a completely new fuzzer to LangFuzzBench. The second situation is modifying the build process of a fuzzer that has been compiled at least once before on the same system. Both situations should bring considerable time savings compared to the baseline (compiling the whole target-fuzzer integration from scratch), however, the former situation will be a bit more costly than the latter because **the name** of the fuzzer influences the instrumentation phase of the target.

We pick *Hermes* as the invariable target for these experiments and consider 7 different grammar-aware fuzzers. We start by first measuring the baselines, which is time needed to compile a benchmark for fuzzing Hermes with each of the fuzzers completely from scratch (on a system with a completely empty Nix store). This is achieved by simply clearing the Nix store after each target-fuzzer integration build. The baseline results are shown as blue bars on Figures 5.1 and 5.3. For evaluating the situation of adding a new fuzzer, we simply build Hermes-Fuzzer integrations in the order from left to right in Figure 5.1, without the previous step of emptying the Nix store when moving to the next integration. What this means is that many components of the benchmark need not be rebuilt. These include target (*Hermes*) compilation, toolchain (*AFL++*) compilation, Clang wrapper compilation etc. The time savings observable in Figure 5.1 appear to be significant. On average, the required build time is 3 times lower than the baseline. We also measured the number of Nix derivations that need to be rebuilt (Figure 5.2).

For the situation of changing the fuzzer, the experiment again proceeds from left to right with respect to the fuzzers from Figure 5.3. We first measure the baseline for a particular integration. Then, we add a dummy OS command in the *prePatch* phase of the Nix script for building the fuzzer and rebuild the same integration. This will change the hash of the fuzzer derivation and force Nix to recompile it. In this manner, we simplify but still mimic actual modifications to the fuzzer a developer would make, such as for example modifying the grammar provided at build-time. Before moving on to the next integration (next pair of adjacent bars in Figure 5.3), we clear the entire Nix store. We then repeat the process for the next integration. The build time improvements shown in Figure 5.3 appear to be even bigger than in the previous experiments (30 times faster than baseline on average). This is because the instrumentation of the target does not need to be executed again, because the name of the fuzzer and the instrumentation logic remain the same. Therefore, in cases of changing the fuzzer build logic, by for example modifying the grammar passed to the fuzzer at compile-time, Nix will trigger neither the recompilation nor the reinstrumentation of the target. Naturally, the toolchain and the Clang wrapper do not need to be recompiled as well. Indeed, the number of Nix derivations that need to be built (Figure 5.4) is even smaller than in the case of adding a new fuzzer (Figure 5.2), as we do not need to reinstrument the target now.

We now execute these two experiments on Magma. Magma is not specialized in grammar-aware fuzzing. Therefore, it is difficult to directly map the notion of the fuzzer from LangFuzzBench to a

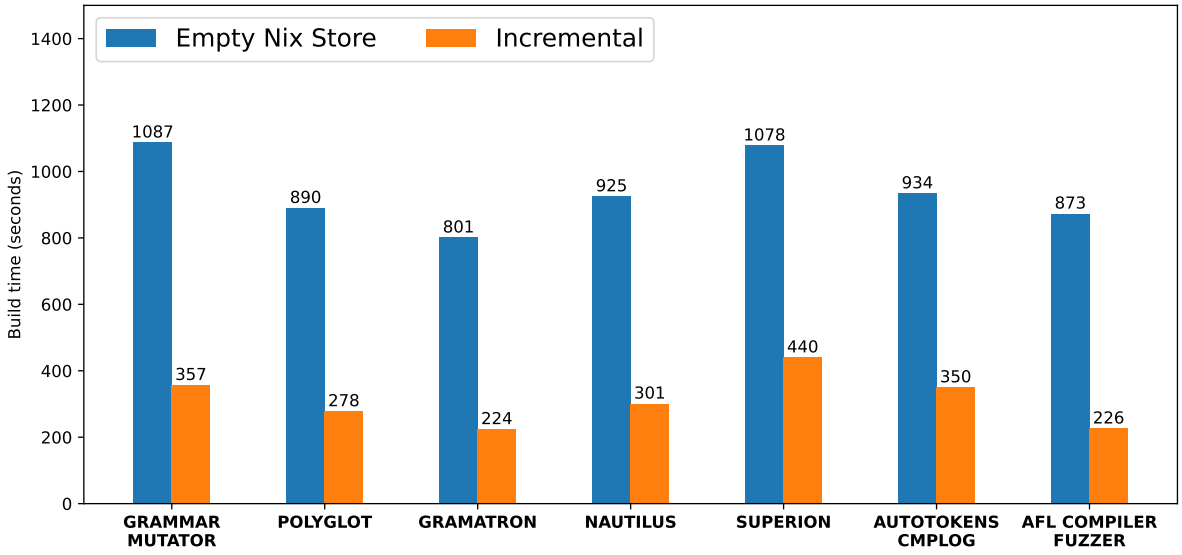

Figure 5.1 – *Average time in seconds required to build a benchmark with Hermes and the given fuzzer. The blue bars show the baseline times, which is the average time required to build the benchmark from scratch. The orange bars display the average times needed to build the same benchmark on a system on which a benchmark for Hermes has already been compiled with another fuzzer from previous iterations of the experiment. The results show an average build-time reduction of more than 3 times.*

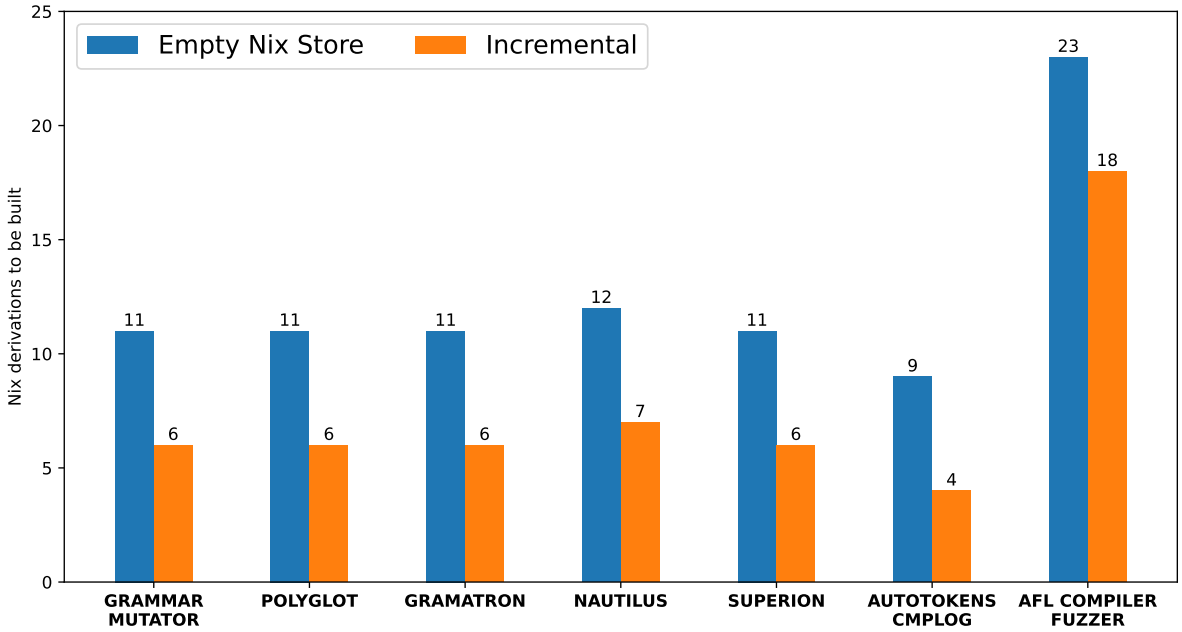

Figure 5.2 – *Results of the same experiment presented in Figure 5.1, but this time we display the number of Nix derivations that had to be built. For each fuzzer we observe a reduction of 5 derivations that need to be built. One is for the Clang wrapper, another for the AFL++ toolchain, another for building Hermes and two more helper derivations used by Nix.*

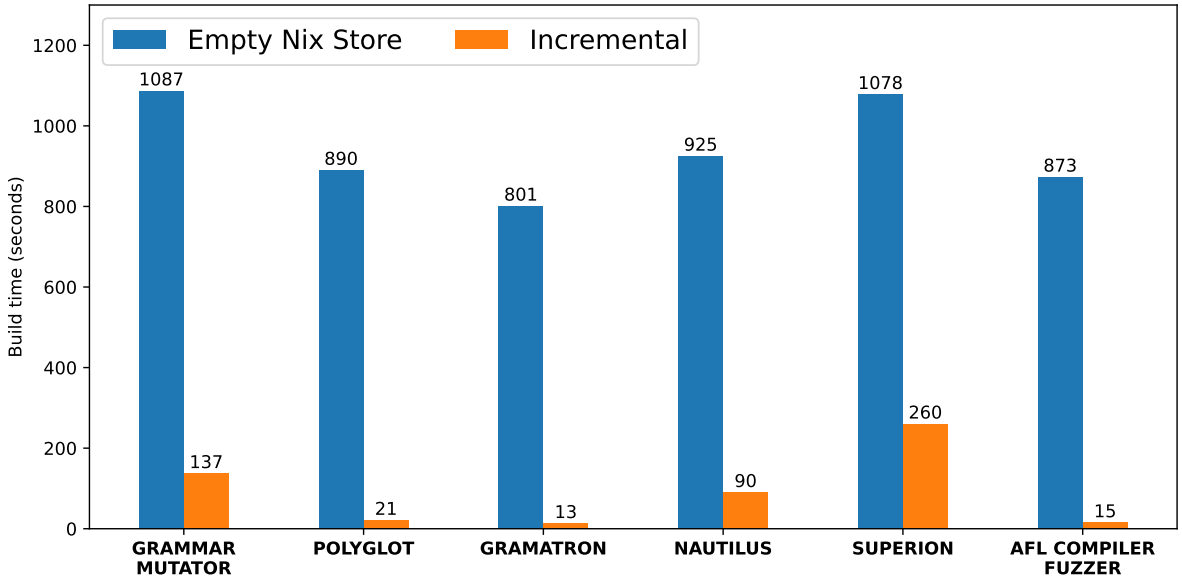

Figure 5.3 – *Average time in seconds required to build a benchmark with Hermes and the given fuzzer. The blue bars show the baseline times, which is the average time required to build the benchmark from scratch. The orange bars display the times needed to rebuild the same benchmark after modifying the logic of building the fuzzer and without emptying the Nix store or cache. We observe an average build-time reduction of around 30 times on average across the considered fuzzers. This is greater than in Figure 5.1 as we do not need to reinstrument the target (Hermes).*

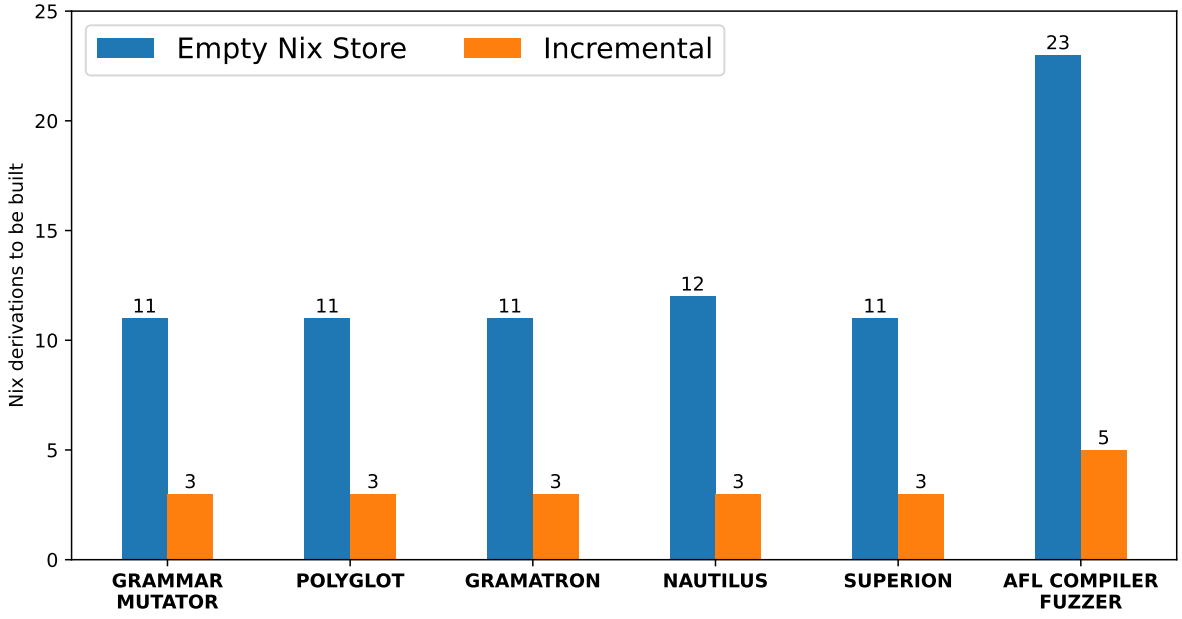

Figure 5.4 – *Results of the same experiment presented in Figure 5.3, but this time we display the number of Nix derivations that had to be built. The reduction of the number of required derivations is greater than in Figure 5.2 as we do not need to reinstrument the target (Hermes).*

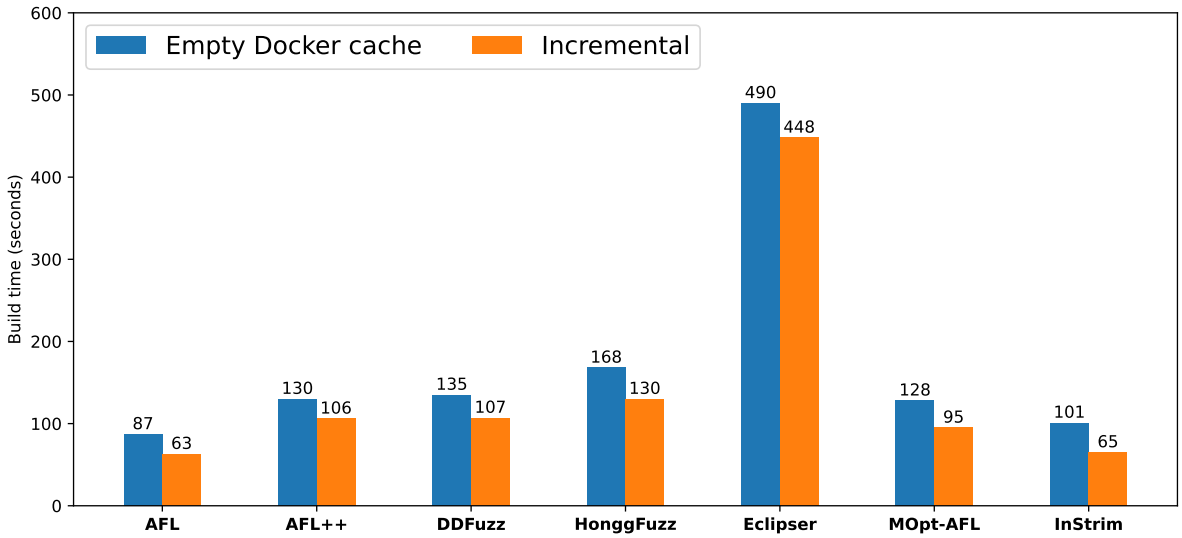

Figure 5.5 – *Average time in seconds required to build a Magma benchmark with Lua and the given fuzzer. The blue bars show the baseline times, which is the average time required to build the benchmark from scratch. The orange bars display the average times needed to rebuild the same benchmark after modifying the logic of building the fuzzer and without emptying the Docker cache. We observe a build-time reduction mainly driven by Docker caching the Magma utility. The target still needs to be both recompiled and reinstrumented. Reinstrumentation may be actually necessary, however, recompilation of the target is certainly not. The highlights of LangFuzzBench's loosely-coupled design and using Nix are best illustrated in Figures 5.1 and 5.3.*

component in Magma. In reality, the fuzzer component in Magma corresponds to both the fuzzer and the toolchain components in LangFuzzBench. Furthermore, Magma does not integrate the same set of targets as LangFuzzBench. Therefore, we will not be comparing the absolute times required to build benchmarks in the two frameworks, but will compare the degree of reduction of build-times when compiling incrementally, as explained in the previous two experiments. We pick the Lua target as well as AFL, AFL++, DDFuzz [21], HonggFuzz [13], Eclipser [9], MOpt-AFL [20] and InsTrim [18] fuzzers for measurements. The results of both experiments are shown in Figure 5.5. In Magma, the build times for compiling a benchmark for fuzzing a particular target are identical regardless of whether the considered fuzzer has been used for the first time with the same target (Docker cache empty) or if the considered fuzzer has just been modified (Docker cache populated). For this reason we display the results of both experiments in a single table. The results indicate a reduction of build time that is mainly driven by Docker caching the Magma utility. This happens because this utility is specified before the fuzzer in the Dockerfile [11] used by Magma to build the benchmark. However, as the instructions for building the target are specified after the fuzzer in the Dockerfile, and as Magma does not decouple target compilation from target instrumentation, the target will need to be both recompiled and reinstrumented when adding or modifying the fuzzer.

5.1.2 Target build analysis

An equally important use-case for LangFuzzBench is the addition of new targets, or modifications of previously built and fuzzed targets. In both of these cases, the target needs to be both compiled and instrumented again. On the other hand, our hypothesis is that the previously built fuzzer derivations should remain fully reusable when modifying previously compiled targets. However, addition of targets implies addition of grammars, so fuzzers will need to be rebuilt in this case.

We run experiments identical to the ones in the previous section. We use a fixed fuzzer now, Grammar-Mutator, and switch 5 different targets between experiment runs: CPython, Lua, MRuby, Hermes and Yara. We first analyze how the addition of new targets affects the speed of incremental benchmark builds using the Grammar-Mutator fuzzer. Results are displayed in Figure A.1 (average build speed in seconds) and Figure A.2 (number of compiled Nix derivations). We see an average reduction of 1.75 times when considering average build time of the entire benchmark. This is less than the average reduction of 3 times that was observed when performing incremental builds with a fixed target (see Figure 5.1), which can be attributed to the fact that targets are usually bigger projects than fuzzers, hence they require more time to build. However, when we look at the reduction in the number of required Nix derivations from Figure A.2, we again see that 4 derivations get always reused. These are the AFL++ toolchain, the Clang wrapper and two more helper derivations for Nix. Note that the fuzzer needs to be rebuilt, as new targets bring previously unseen grammars that are needed at compile-time by the fuzzer.

When considering the scenario in which the benchmark is compiled incrementally after modifying a previously compiled target, performance benefits are again bigger, as indicated in Figures A.3 and A.4. This is because in addition to all the derivations that got reused in the case of adding a new target, the fuzzer derivations will now also be reused. In this case, the fuzzer is completely decoupled from the target.

RQ1 answer: LangFuzzBench offers considerable build time savings when recompiling benchmarks. These savings are driven by the Nix build system and the clear separation of target compilation and instrumentation phases. It offers greater time savings than Docker-based fuzzing frameworks like Magma. This design benefits not just the users of the framework, but also those who wish to offer a fuzzing platform as a service (like Fuzzbench).

5.2 RQ2: Performance of grammar-aware fuzzers

In this section we will focus on questions **RQ2**. In fuzzing, there is often a gap between the ideas described in research papers and what is actually implemented in publicly available prototypes. One of the often overlooked facts is whether the grammar-aware fuzzer employs traditional byte-level

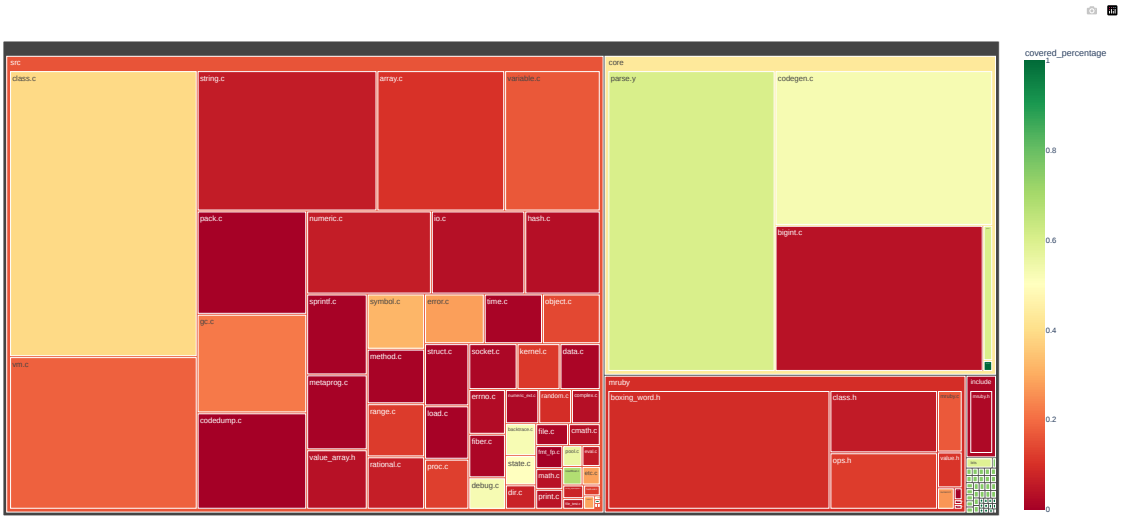

Figure 5.6 – Edge coverage per module of MRuby when fuzzed with vanilla AFL++. The modules in the src folder (left part of the image) correspond to "deeper" code that cannot be easily achieved with purely *byte-level mutational* operators, like the one employed by AFL++.

mutations from AFL and AFL++ in addition to all the grammar-aware operators introduced by the paper. This becomes a very important detail if we consider that *byte-level mutational* operators are best suited for finding bugs that are in the target's frontend, whereas grammar-aware mutations work best for bugs with root cause in the target's backend. This relationship is perfectly illustrated in Figures 5.6 and 5.7, which illustrate the edge coverage we obtained while fuzzing MRuby with vanilla AFL++ and Nautilus respectively. Not surprisingly, AFL++ dominated in modules related to lexing and parsing, whereas Nautilus achieved much better edge coverage in "deeper" code. We illustrate the same trade-off in results of fuzzing Yara with vanilla AFL++ and Grammar-Mutator in Table A.1. We also note that some of the fuzzers, even though they are implemented on top of AFL++ custom mutator API, simply cannot run in conjunction with AFL++ byte level mutations. This is because they have not been hardened for mutating inputs that do not comply with the provided grammar. Fuzzers that fall into this category are Grammar-Mutator and Gramatron.

We now present the delta edge coverage in Table 5.1 and number of bugs discovered in Table 5.2 during the campaigns we run for all fuzzer-target pairs that have been integrated into LangFuzzBench. Each campaign ran for 24 hours and was repeated 5 times before computing the average edge coverage and standard deviation. For grammar-aware fuzzers, we now only show results for campaigns that used the most optimized grammars for that particular fuzzer. Fuzzers that can be configured whether or not to use byte-level mutations in addition to grammar-aware operators are included twice (Polyglot and Autotokens). When running Autotokens in purely grammar-aware mode, we had to turn off CMPLOG instrumentation (it was not supported). Also, note that AFL-Compiler-Fuzzer and Superion run on top of AFL (not AFL++), which can explain their lower

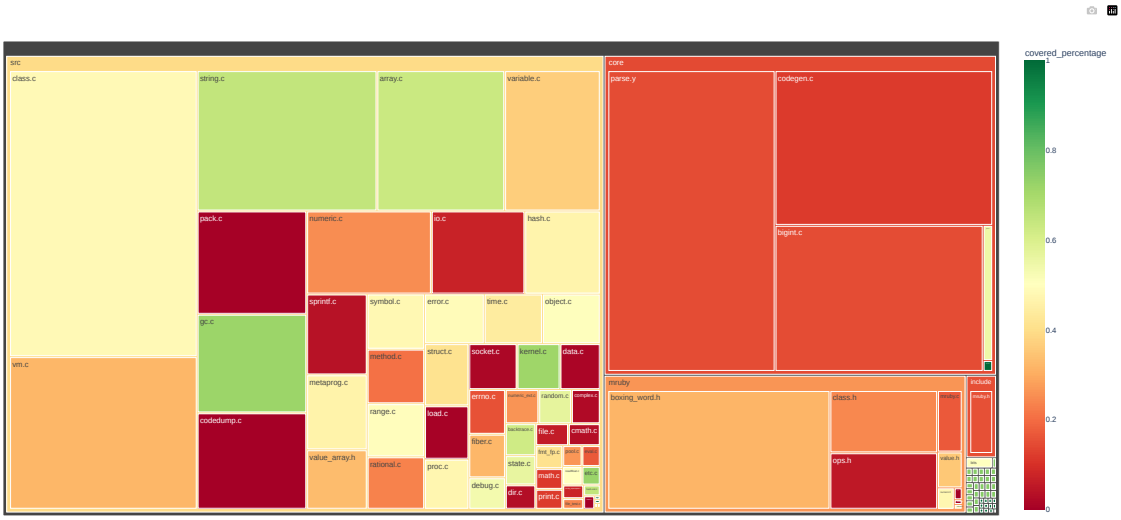

Figure 5.7 – Edge coverage per module of MRuby when fuzzed with Nautilus. The modules in the core folder (upper right part of the image) correspond to the "frontend" code that cannot be easily achieved with purely grammar-abiding mutational operators, like the ones used by Nautilus.

performance.

Comparing performances of different grammar-aware fuzzers is difficult because it is impossible to decouple the impact of the provided grammar and corpus from the impact of the mutation operators and their scheduling. Comparing delta edge coverage indeed eliminates trivial coverage from the corpus, but better corpus will still influence the course of the campaign. However, we can easily draw conclusions from comparing Polyglot(POL) with Polyglot-AFL++(POL-A++), or Autotokens(AUT) with Autotokens-AFL++(AUT-A++), as these pairs of fuzzers begin fuzzing from the same starting position. From Table 5.1, we observe that out of 30 fuzzer-target integrations, purely grammar-aware fuzzers obtained greater edge coverage than their hybrid counterparts on only 5 integrations (CRuby-Autotokens, Luau-Autotokens, Hermes-Autotokens, JavaScriptcore-Autotokens and Solidity-Polyglot). Likewise, AFL++(A++) outperformed Polyglot-AFL++(POL-A++) only on 3 integrations (PHP, Solidity and Lua), whereas for Autotokens-AFL++ it fell behind on every target we fuzzed. If we consider the number of discovered bugs (Table 5.2), we see that Autotokens(AUT) discovered 0 bugs, whereas Autotokens-AFL++(AUT-A++) discovered 20. Polyglot(POL) discovered 2 bugs, whereas Polyglot-AFL++(POL-A++) discovered 3, including both of those discovered by plain Polyglot(POL). On the other hand, vanilla AFL++(A++) discovered 8 bugs, 5 of which were also discovered by Autotokens-AFL+(AUT-A++). Our main takeaway from these results is that purely grammar-aware fuzzers will struggle to find coverage and bugs that simply cannot be expressed through grammar-abiding mutations. We illustrate this point again through Figure 5.8.

	G1	G2: Pure grammar-aware fuzzers					G3: Hybrid fuzzers			
Target	A++	GRA	NAU	G-MUT	AUT	POL	AUT-A++	POL-A++	ACF	SUP
libcsv	3.46	**6.67**	1.38	1.09	3.2	0.43	3.46	3.46	0.11	3.03
libxml	2.92	2.58	1.28	0.46	5.76	0.21	**6.57**	3.99	2.61	2.53
jq	1.39	1.21	1.18	**2.44**	0.91	0.17	1.41	1.41	1.26	1.28
mruby	12.82	**27.91**	19.03	13.69	18.67	3.89	19.9	19.25	5.34	6.12
cruby	2.7	/	**7.08**	3.81	5.25	1.11	2.9	5.48	0.3	/
lua	21.2	21.77	**41.61**	31.05	23.25	9.62	27.47	20.92	18.71	20.32
luau	5.57	5.63	11.02	8.75	**12.13**	5.44	7.5	7.69	4.8	6.05
cpython	2.78	2.27	**6.52**	4.48	2.59	1.65	3.7	4.0	2.71	2.45
mpython*	1.06	4.67	**11.82**	10.59	0.12	0.48	1.25	1.83	0.79	0.37
hermes	1.44	15.78	**17.56**	/	13.91	1.26	5.45	2.33	1.0	1.3
JSCore*	0.6	/	**10.06**	/	3.17	0.64	1.33	0.66	0.54	0.73
sqlite	11.25	6.95	**19.41**	11.55	12.71	9.23	15.16	13.05	6.82	8.29
yara	7.03	7.55	11.28	**12.44**	7.04	3.96	10.03	8.79	6.85	8.45
solidity	0.61	1.06	**4.27**	3.39	0.63	1.26	0.75	0.56	0.39	0.34
php	3.9	**11.6**	8.45	6.23	1.62	1.14	7.14	3.85	2.01	2.07

Table 5.1 – *Average delta edge coverage in percentages (achieved edge coverage without the corpus) against all targets and 10 different fuzzer setups: AFL++(A++, baseline), Gramatron(GRA), Nautilus(NAU), Grammar-Mutator(G-MUT), Autotokens (AUT), Polyglot(POL), Autotokens with CMPLOG and AFL++(AUT-A++), Polyglot with AFL++(POL-A++), AFL compiler fuzzer(ACF) and Superion(SUP). mpython* and JSCore* stands for micropython and javascriptcore, respectively. Results of purely byte-level fuzzers are represented in column group G1, purely grammar-abiding fuzzers in column group G2, whereas those using both types of mutations in column group G3. Best performing integrations are represented in bold.*

```
1   pragma experimental solidity;
2   //=4/t C {
3       /// @irBct C {s addre?s coin;
4        type c= 2|4: (,8-1): Modi2;
5   //{
6       /// @irBct C {s add2.X = 1; p2.Y
7
8
9
10
```

Figure 5.8 – PoC to trigger Solidity bug 2 from Table 5.2. This bug was only discovered by vanilla AFL++, as well as the two hybrid fuzzers that combine byte-level and grammar-aware mutations - Polyglot-AFL++ and Autotokens-AFL++. Purely grammar-aware fuzzers will struggle to find bugs that cannot be expressed by a grammar.

	G1	G2: Pure grammar-aware fuzzers					G3: Hybrid fuzzers			
Target	A++	GRA	NAU	G-MUT	AUT	POL	**AUT-A++**	POL-A++	ACF	SUP
mruby 1		X		X						
yara 1									X	X
hermes 1		X	X							
hermes 2		X				X		X		
solidity 1	X		X			X		X		
solidity 2	X						X	X		
lua 1							X			
lua 2							X			
lua 3							X			
lua 4							X			
lua 5							X			
mpython 1			X	X					X	
mpython 2			X							
mpython 3			X	X			X			
mpython 4			X							
mpython 5	X						X			
mpython 6							X			
mpython 7							X			
mpython 8							X		X	
mpython 9							X			
mpython 10							X			
mpython 11							X			
mpython 12							X			
mpython 13	X						X			
mpython 14							X			
mpython 15							X			
mpython 16	X						X			
mpython 17	X						X			
mpython 18									X	X
mpython 19										X
mpython 20	X									
mpython 21	X									

Table 5.2 – *Bugs discovered with different fuzzers: AFL++(A++, baseline), Gramatron(GRA), Nautilus(NAU), Grammar-Mutator(G-MUT), Autotokens(AUT), Polyglot(POL), Autotokens with CMPLOG and AFL++(AUT-A++), Polyglot with AFL++(POL-A++), AFL compiler fuzzer(ACF) and Superion(SUP). Results of purely byte-level fuzzers are represented in column group G1, purely grammar-abiding fuzzers in column froup G2, whereas those using both types of mutations in column group G3. The best performing fuzzer, Autotokens with CMPLOG and AFL++, is marked in bold.*

RQ2 answer: Hybrid mutational fuzzer setups that perform both byte-level and grammar-abiding mutations outperform setups of the same fuzzer that simply perform grammar-abiding mutations. Fuzzers that mutate the input queue exclusively based on a grammar will struggle to find bugs that cannot be reached with a test case generated by such a grammar.

5.3 RQ3: Grammar design strategies for fuzzing

We analyze question **RQ3** first from the perspective of corpora-free generators - Nautilus and Grammar-Mutator. We develop two different grammars for each target fuzzed by these two fuzzers. The first grammar is a translation of an ANTLR grammar found on the home page of the target or the GitHub repository [6] that provides a collection of ANTLR grammars used for **parsing**. We try to stick to the original ANTLR grammar as much as possible. We also manually add whitespaces where necessary in order to help the fuzzers get through at least lexical analysis, but we do not add any vocabulary (built-in function names, module names, global variables etc.). The second grammar is an extension to the first one that includes the vocabulary which was previously missing. Table 5.3 shows the resulting delta edge coverage. Since Nautilus generates 100 test cases to act as the corpus, we compute the delta edge coverage by simply ignoring the first 100 elements from the queue with the lowest timestamp. Slightly different from Nautilus, Grammar-Mutator requires users to manually provide the "corpus" that **was generated by the grammar**, so in this case we compute delta edge coverage simply by ignoring this "corpus". From Table 5.3, we can see that the second version of the grammar achieved greater edge coverage on average against almost every target, the exception being Nautilus-SQLite and Grammar-Mutator-Luau integrations. These results suggest that, when using fuzzers that generate test cases from the grammar, adding vocabulary to the grammar can considerably increase edge coverage.

However, using vocabulary-enhanced grammars with corpora-dependent fuzzers, like Polyglot, brings new challenges because of the requirement to adapt the corpus in order to ensure it remains fully parsable. To prove the impact of this, we split the targets into two groups of six and fuzz them with Polyglot. For the first group of targets, we do not make any changes to the corpus, but still use the vocabulary-enhanced grammars. For the second group of targets, we also use the vocabulary-enhanced grammars but adapt the corpus so that it remains fully parsable. Average delta edge coverage results are displayed in Table 5.4. Firstly, we observe that the first group of targets suffered an average reduction in obtained delta edge coverage of 0.85 percentage points, compared to the baseline averages which use the generic ANTLR grammars for parsing. On the other hand, the second group of targets observed and average increase in delta edge coverage of 1.29 percentage points. We notice how this increase is lower than what we observed for corpora-free generators (see Table 5.3). This is because the corpus used for fuzzing targets with Polyglot with baseline grammars already includes a lot of the vocabulary of the language. This is however not the case for corpora-free

	Nautilus		Grammar-Mutator	
Target	Grammar 1	Grammar 2	Grammar 1	Grammar 2
mruby	11.6	19.03	12.07	13.69
cruby	4.3	7.08	2.92	3.81
lua	22.21	41.61	16.71	31.05
luau	8.52	11.02	8.75	7.33
cpython	3.71	6.52	4.46	4.48
mpython	10.86	11.82	9.93	10.59
hermes	8.35	17.56	/	/
JSCore	3.16	10.06	/	/
sqlite	19.41	18.1	11.16	11.55
yara	8.92	11.28	10.17	12.44
solidity	3.65	4.27	3.28	3.39
php	0.19	8.45	0.35	6.23

Table 5.3 – *Average delta edge coverage in percentages (achieved edge coverage without the "corpus") of corpora-free generators against all targets integrated into LangFuzzBench, apart from libcsv, libxml and jq. Grammar 1 is a translated version of the generic grammar used for parsing, whereas Grammar 2 is an improved version of this grammar that keeps most of the production rules but also adds a lot of language vocabulary.*

generators - these fuzzers use the grammar to generate the corpus, so the grammar becomes the only source of vocabulary. Not surprisingly, performance spikes in delta edge coverage when using vocabulary-enhanced grammars are thus greater for corpora-free generators.

RQ3 answer: Enhancing the grammar with vocabulary is an effective way to boost fuzzers performance. It works best for fuzzers that use the grammar to generate the initial corpus. For fuzzers that use a pre-collected corpus, it mandates another preprocessing step.

	Polyglot	
Target	Grammar 1	Grammar 2
mruby	3.89	1.16
cruby	1.11	0.9
lua	9.62	7.92
luau	5.44	5.1
hermes	1.26	1.18
javascriptcore	0.64	0.63
...	...	...
cpython	0.96	1.65
micropython	0.16	0.48
sqlite	4.98	9.23
yara	3.01	3.96
solidity	0.45	1.26
php	0.41	1.14

Table 5.4 – *Average delta edge coverage in percentages (achieved edge coverage without the "corpus") of Polyglot against all targets integrated into LangFuzzBench, apart from libcsv, libxml and jq. Grammar 1 is a translated version of the generic grammar used for parsing, whereas Grammar 2 is an improved version of this grammar that keeps most of the production rules but also adds language vocabulary. For the 6 upper targets (mruby, cruby, lua, luau, hermes and javascriptcore), no corpora adaptations were made in order to ensure it remains parsable by Grammar 2. For the lower 6 targets (cpython, micropython, sqlite, yara, solidity and php), corpus was adjusted so that no test cases got discarded as unparsable by Grammar 2. The upper 6 targets suffered a reduction of delta edge coverage by 0.85 percentage points on average, whereas the lower group of targets observed a spike in delta edge coverage by 1.29 percentage points on average. A percentage point is computed simply as a difference of two percentages.*

Chapter 6

Related Work

6.1 Fuzz benchmarks

In this section we will discuss some of the fuzzing benchmarks that have been popularized in recent years. We will also address notable focal point and design differences between these benchmarks and LangFuzzBench.

6.1.1 Magma

Magma [17] is an open fuzzing benchmark specifically designed with the aim of objective fuzzer evaluation and comparison against a broad set of targets. To achieve this, Magma focuses on developing bug-centric metrics that provide a more insightful feedback on the success of the fuzzer. The intuition is that widely used fuzzer metrics, such as number of triggered crashes, number of discovered bugs after manual crash deduplication, and the achieved coverage profile are not insightful enough when assessing the fuzzers capability to achieve its main task, which is to *find bugs*. Therefore, Magma adds custom instrumentation to vulnerable targets that aims to detect the states of *a) reaching the bug* and *b) actually triggering the bug*.

At the time of writing this, Magma includes 26 different fuzzer profiles and 9 different targets. Currently, no grammar-aware mutational fuzzers are supported by Magma. However, since AFL++ is supported, some of the grammar-aware fuzzers from LangFuzzBench, like Polyglot, Grammar-Mutator and autotokens (Token-Level AFL), could be easily ported to Magma. Furthermore, Magma does not focus only on targets that accept highly structured text inputs. Currently, only 4 targets are integrated to both Magma and LangfuzzBench (PHP, Lua, SQLite and libxml2).

A notable design distinction between Magma and LangFuzzBench is the use of Docker and

Nix. Magma, which uses Docker to build a separate container for each fuzzer-target integration, maximizes caching by careful scheduling of build instructions in the Docker file. However, this still results in tight coupling of fuzzers and targets, as shown in the evaluation section of our paper. Conversely, LangFuzzBench relies on the Nix package manager, which results in modular derivations that can be reused irrespectively of their position in the configuration file. Furthermore, Magma does not decouple the compilation and instrumentation phases of the target build phase. On the other hand, LangFuzzBench utilizes the LLVM link time optimization to separate target instrumentation from target compilation phase, which further amplifies time savings when evaluating the same target against multiple fuzzers.

6.1.2 Fuzzbench

Fuzzbench [22] is an open-source platform developed by Google that offers a free service for evaluating fuzzers. Fuzzbench aims to foster research in the field of fuzzing by providing not only a framework for seamless integration of new fuzzers, but also computational resources needed to run the fuzzing campaigns. It also offers a reporting toolkit with detailed metrics overview and statistical tests that alleviate interpreting the results of the campaign and the performance of the fuzzer. Fuzzbench has had a major impact on the security research community and has become the de facto standardized platform for fuzzer benchmarking. For example, OSS-Fuzz [26] migrated from AFL [1] to AFL++ [12] due to the results the latter fuzzer demonstrated in Fuzzbench [27].

At the time of writing this, there are more than 70 fuzzers and 28 benchmarks (targets) integrated in the Fuzzbench project. Unlike Magma, Fuzzbench already integrates some of the fuzzers also present in Langfuzzbench, such as Nautilus, Gramatron and Autotokens (Token-Level AFL). There are also 3 targets present in both Fuzzbench and Langfuzzbench: PHP, SQLite and libxml2. Core distinctions in benchmark design between LangFuzzBench and Fuzzbench are identical to those already addressed in the section dedicated to Magma. The only difference is that Fuzzbench was not designed for running local experiments (although such an option now exists). Nevertheless, fuzzer developers are still expected to make sure their fuzzers can be compiled by the framework, so local fuzzer builds and prototyping are expected by the developers. Therefore, reusable design and build efficiency of the benchmark are still relevant. It is actually even more relevant if we consider that Fuzzbench is a service offered globally, with presumably hundreds of campaigns running daily.

6.2 Grammar-aware fuzzers

In this section, we will discuss the core ideas behind different grammar-aware mutational fuzzers that have been integrated into LangFuzzBench.

6.2.1 Polyglot

Polyglot [8] is a grammar-aware mutational fuzzer that aims to neutralize the differences between syntax and semantics by utilizing a uniform intermediate representation (IR). Polyglot was designed to address the lack of semantic correctness when using grammars originally written for parsing to generate actual test cases. This is achieved by employing various semantic validations when mutating test cases. These validations can be customized by the user in a separate configuration file before starting the campaign. However, the version of Polyglot [28] that has been integrated in LangFuzzBench currently does not support many types of semantic validations, as they are still in the process of being ported from the older Polyglot project [29]. Therefore, Polyglot is expected to find similar classes of bugs as rest of the grammar-aware mutational fuzzers integrated in LangFuzzBench. The new version of Polyglot has been implemented on top of AFL++ custom mutator API.

6.2.2 Superion

Superion [32] is one of the first grammar-aware mutational fuzzers to demonstrate tangible benefits of using grammar-centric fuzzing approaches. It was designed to work on top of AFL an addition of all the byte-level mutations performed by AFL. Superion introduced two new mutation strategies. Firstly, it enhances AFL's dictionary-based approach by overwriting tokens in a grammar-aware manner. Furthermore, it performs tree-based mutations by replacing one subtree of the AST with another subtree that can originate from the same test case or any other input from the queue. It succeeded to find 34 new bugs and get 19 CVEs. However, Superion has only been evaluated against JavaScript (WebKit, Jerryscript and ChakraCore) and XML (libplist) based targets.

6.2.3 Nautilus

Nautilus [7] is one of the first fuzzers to combine the grammar-centric mutation approaches with coverage-guided fuzzing. It has been evaluated on ChakraCore, PHP, MRuby and Lua and discovered 6 CVEs. The main selling point of Nautilus compared to the majority of the other fuzzers that have been integrated to LangFuzzBench is that it does not require a corpus in order to fuzz a target. When the campaign starts, Nautilus will use the provided Grammar to generate the specified number of test cases (in our experiments 100) that will serve as corpora. This puts even greater importance on grammar design. In order to address the bias factor introduced by the grammar, Nautilus tries to generate test cases in a uniform manner by prioritizing rules of the grammar that can generate more different subtrees. A rule that can generate 4 different subtrees will be picked twice as often as the rule that can generate only 2 different subtrees. Nautilus also uses various minimization techniques in order to generate smaller samples that trigger the same coverage. This has many benefits on execution speed, by for example leaving a smaller set of mutations to work on in the future. It also supports grammars that are not context-free by introducing custom scripting support

when defining grammars that will facilitate producing semantically valid test cases.

6.2.4 Grammar-Mutator

Grammar-Mutator [4] is a grammar-aware mutator implemented on top of AFL++ custom mutator API. It combines different ideas from F1 fuzzer [14] and Nautilus, but takes most of the mutator operators from the latter. Similarly to Nautilus, it does not require a corpus collected in advance. Conversely, it uses a grammar specified in JSON format to generate seeds and the corresponding trees. Although implemented on top of AFL++ custom mutator API, currently it is possible to run Grammar-Mutator only in the AFL_CUSTOM_MUTATOR_ONLY mode, which means that no AFL++ mutations (byte-level mutations) will be applied.

6.2.5 Gramatron

Gramatron [31] uses grammar automatons to guide grammar-aware fuzzing. Grammar automatons are grammars converted to finite state automatons (FSA). Gramatron uses FSAs to address the performance overhead introduced by parse tree representations. The first goal of Gramatron was to eliminate sampling bias that arises from generating test cases from grammars. The second goal of Gramatron was to escape futile, localized searches that arise due to applying only small mutations to the input samples. To achieve this, Gramatron uses more aggressive mutation operators. Gramatron also provides an integration with AFL++ custom mutator API, although in LangFuzzBench we have integrated the original version [15]. Similarly to Grammat-Mutator, Gramator currently supports only AFL_CUSTOM_MUTATOR_ONLY mode, which means we could not run it in addition to AFL++ byte-level mutations.

6.2.6 Token-Level AFL

Token-Level AFL [30] adopts an approach between byte-level fuzzers and grammar-aware fuzzers. It does not maintain any explicit notion of a grammar, but it does use a set of tokens (a dictionary) that have a reserved meaning in the target's language. Then, when mutating test cases, it does not operate on the byte-level, but on the token-level instead. This means that entire tokens get added, removed or replace from the test cases. This allows the fuzzer to avoid generating inputs that do not even pass the lexical analysis of the underlying target. Ultimately, Token-Level AFL reaches deeper code compared to purely byte-level fuzzers, without paying the full cost of maintaining a full grammar-related data structures. Also, it does not suffer from bias problems that arise from generating test cases from human-written grammars, but it does require a versatile corpus and a comprehensive dictionary. Token-Level AFL is also implemented as a mutator on top of AFL++ custom mutator API under the name *autotokens*.

6.2.7 AFL Compiler Fuzzer

AFL Compiler Fuzzer [16] [2] is an extension implemented on top of AFL and specifically designed for fuzzing targets that expect C-like language files as input (hence the name compiler fuzzer). It has had great success in fuzzing the Solidity compiler. AFL Compiler Fuzzer prioritizes fast and performant mutation operators over smart mutation operators that take long to finish due to the expensive analysis of the IR. Therefore, the core mutation operators of AFL Compiler Fuzzer are basically hard-coded string "search-and-replace" operators, completely agnostic of the target being fuzzed. For example, one mutation operator might search for "!=" and replace it with "==". Other operators might be a bit more involved, such as swapping function argument positions, but they are still designed to have very low overhead. To achieve more sophisticated mutations, AFL Compiler Fuzzer also implements a preprocessing step using the Comby-Decomposer [10] tool that scans the entire corpus and extracts reusable fragments of the code. The rules of how decomposition should be performed are customizable by the user. For example, typical rules would be to extract parts of the code occurring inside (...), {..} and [...] delimiters, in order to reuse previously observed function call arguments, function definition blocks, and array indices during mutations. If the target does not adhere to C-style syntax, decomposition can be customized by the user. AFL Compiler Fuzzer is implemented as an additional step in AFL's havoc phase, which means AFL's byte-level mutators are still used.

Chapter 7

Conclusion

To summarize, in this paper we introduce LangFuzzBench - an extensible fuzzing framework for benchmarking grammar-aware fuzzers. Benchmarks built through LangFuzzBench consist of several loosely-coupled components that can, in most cases, be easily recompiled without requiring recompilation of other components from the same benchmark. Most notably, we split the task of compiling the fuzz target into two distinct components - target compilation and target instrumentation. The former component remains completely independent of the choice of the grammar-aware fuzzer. This means that modifying or adding new grammar-aware fuzzers to LangFuzzBench will not trigger the process of compiling the target if this was already done once on the same system. This offers significant time savings given that many targets that accept highly structured text input, like JavaScript engines, contain a huge codebase that is very time-consuming to compile. We prove that, in cases of incremental recompilation, our design offers greater reductions in benchmark build time compared to some other fuzz frameworks like Magma and Fuzzbench, which rely solely on Docker's caching logic to drive build time reductions. Furthermore, the use of Nix package manager as the foundation for building components of our benchmarks enables a robust, reliable and reproducible build process. This decreases the feedback loop and allows security researchers to focus on their actual tasks, rather than wasting time troubleshooting broken benchmark builds. We use these features of LangFuzzBench to fuzz 150 fuzzer-target integrations (10 fuzzer setups and 15 targets). The results of these campaigns show that purely grammar-abiding fuzz approaches can easily get stuck in localized searches due to the bias introduced through human-written grammars. Certain bugs simply cannot be triggered through grammar-abiding test cases. To alleviate the impact of this, we show how hybrid fuzzers that combine both byte-level and grammar-aware mutations achieve better results in both bug-centric and code-centric metrics. Finally, we discuss different strategies on how to improve the quality of the grammars used for fuzzing. We show that we can achieve much better results when enhancing these grammars with language vocabulary, but also demonstrate how this requires corpus adaptations in case the fuzzer does not use the grammar to generate the initial set of test cases.

Bibliography

[1] *AFL - American Fuzzy Lop.* URL: https://github.com/google/AFL.

[2] *AFL Compiler Fuzzer.* URL: https://github.com/agroce/afl-compiler-fuzzer.

[3] *AFL++ custom mutator API.* URL: https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators.

[4] *AFL++ Grammar Mutator.* URL: https://github.com/AFLplusplus/Grammar-Mutator.

[5] *ANTLR - ANother Tool for Language Recognition.* URL: https://www.antlr.org/.

[6] *ANTLR v4 Grammars - Collection of formal grammars written for ANTLR v4.* URL: https://github.com/antlr/grammars-v4/tree/master.

[7] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. "NAUTILUS: Fishing for Deep Bugs with Grammars." In: *NDSS*. 2019.

[8] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. "One engine to fuzz'em all: Generic language processor testing with semantic validation". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 642–658.

[9] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. "Grey-box Concolic Testing on Binary Code". In: *Proceedings of the International Conference on Software Engineering*. 2019, pp. 736–747.

[10] *Comby-Decomposer - Decompose program automatically into a set of fragments.* URL: https://github.com/comby-tools/comby-decomposer.

[11] *Dockerfile for building benchmarks with Magma.* URL: https://github.com/HexHive/magma/blob/v1.2/docker/Dockerfile.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "{AFL++}: Combining incremental steps of fuzzing research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. 2020.

[13] *Google HonggFuzz - GitHub repository.* URL: https://github.com/google/honggfuzz.

[14] Rahul Gopinath and Andreas Zeller. "Building fast fuzzers". In: *arXiv preprint arXiv:1911.07707* (2019).

[15] *Gramatron (original repository).* URL: https://github.com/HexHive/Gramatron.

[16] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. "Making no-fuss compiler fuzzing effective". In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 2022, pp. 194–204.

[17] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A ground-truth fuzzing benchmark". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (2020), pp. 1–29.

[18] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. "Instrim: Lightweight instrumentation for coverage-guided fuzzing". In: *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*. Vol. 40. 2018.

[19] *LLVM Link Time Optimization: Design and Implementation.* URL: https://www.llvm.org/docs/LinkTimeOptimization.html.

[20] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. "MOPT: Optimized Mutation Scheduling for Fuzzers". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

[21] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. "Fuzzing with data dependency information". In: *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2022, pp. 286–302.

[22] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. "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*. 2021, pp. 1393–1403.

[23] *Nix and NixOS - Declarative builds and deployments.* URL: https://nixos.org/.

[24] *Nix Packages - A repository of more than 100000 packages.* URL: https://search.nixos.org/packages.

[25] *Nix Pills: callpackage design pattern.* URL: https://nixos.org/guides/nix-pills/13-callpackage-design-pattern.

[26] *OSS-Fuzz - Continuous fuzzing for open-source software.* URL: https://github.com/google/oss-fuzz.

[27] *OSS-Fuzz GitHub commit - Migrating from AFL to AFL++.* URL: https://github.com/google/oss-fuzz/issues/4280.

[28] *PolyGlot, a fuzzing framework for language processors (new version).* URL: https://github.com/OMH4ck/PolyGlot.

[29] *PolyGlot, a fuzzing framework for language processors (old version).* URL: https://github.com/s3team/Polyglot.

[30] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. "{Token-Level} Fuzzing". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2795–2809.

[31] Prashast Srivastava and Mathias Payer. "Gramatron: Effective grammar-aware fuzzing". In: *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 2021, pp. 244–256.

[32] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. "Superion: Grammar-aware greybox fuzzing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 724–735.

Appendix A

Evaluation Figures

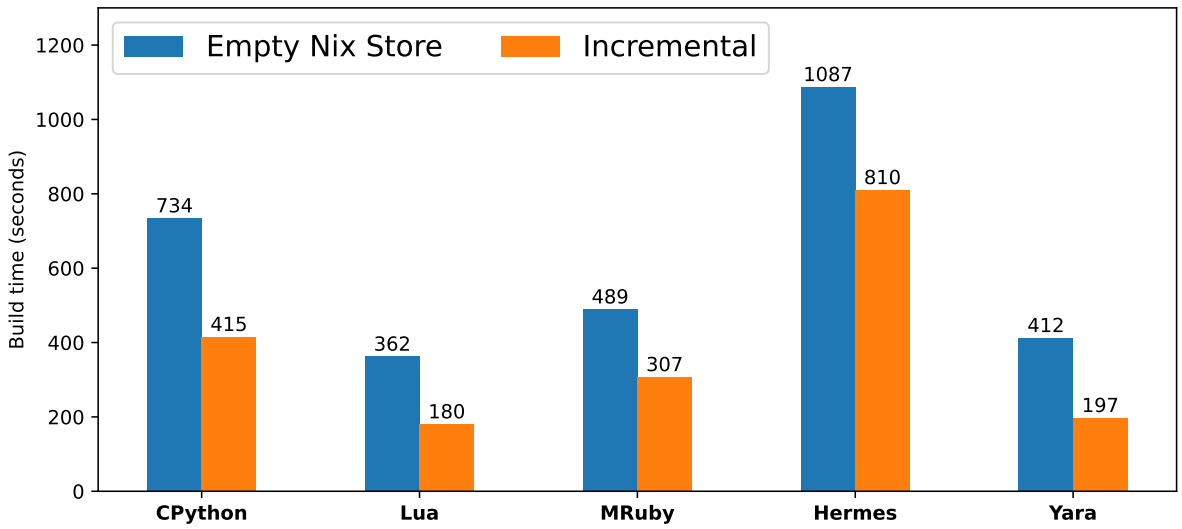

Figure A.1 – *Average time in seconds required to build a benchmark with Grammar-Mutator and the given target. The blue bars show the baseline times, which is the average time required to build the benchmark from scratch. The orange bars display the average times needed to build the same benchmark on a system on which a benchmark for Grammar-Mutator has already been compiled with another target from previous iterations of the experiment. The results show an average build-time reduction of more than 1.75 times.*

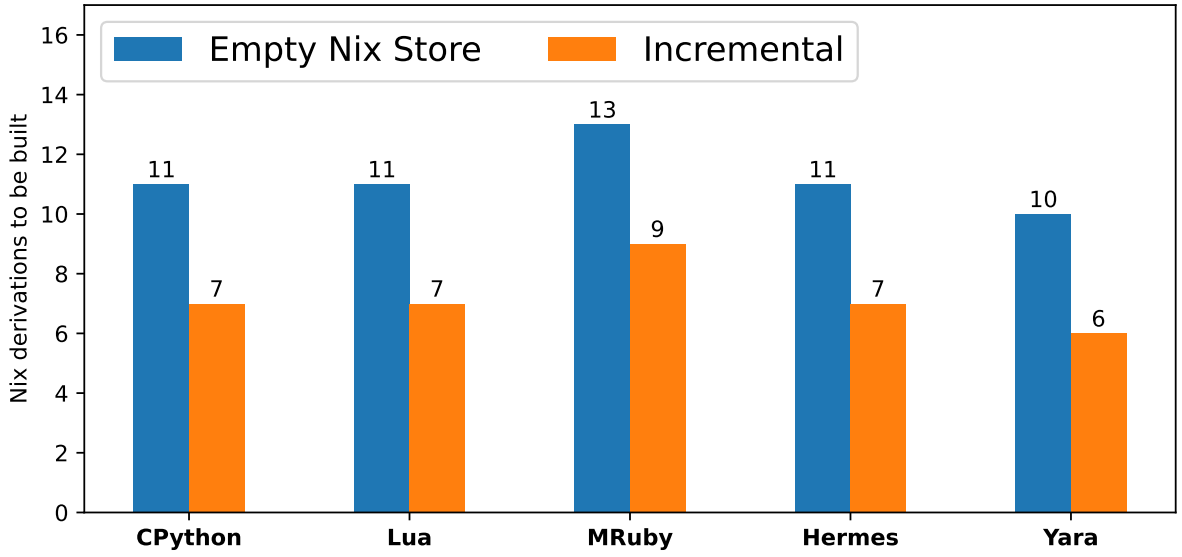

Figure A.2 – *Results of the same experiment presented in Figure A.1, but this time we display the number of Nix derivations that had to be built. For each target we observe a reduction of 4 derivations that need to be built.*

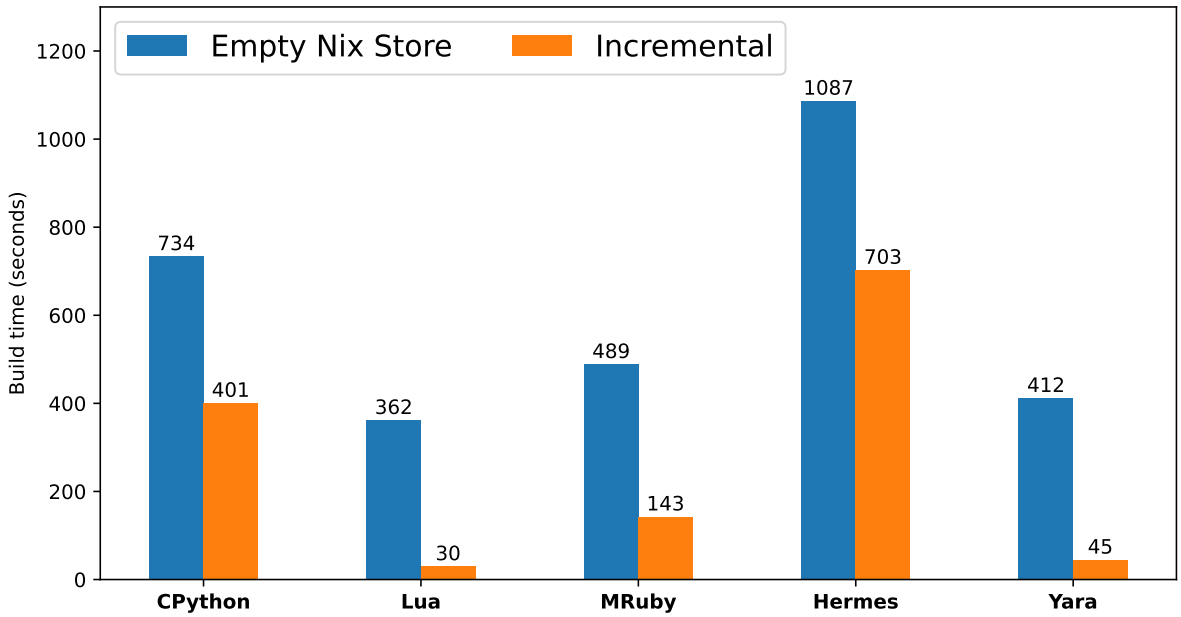

Figure A.3 – *Average time in seconds required to build a benchmark with Grammar-Mutator and the given target. The blue bars show the baseline times, which is the average time required to build the benchmark from scratch. The orange bars display the times needed to rebuild the same benchmark after modifying the logic of building the target and without emptying the Nix store or cache. We observe an average build-time reduction of around 5.6 times on average across the considered targets. This is greater than in Figure A.1 as we do not need to rebuild the fuzzer (Grammar-mutator).*

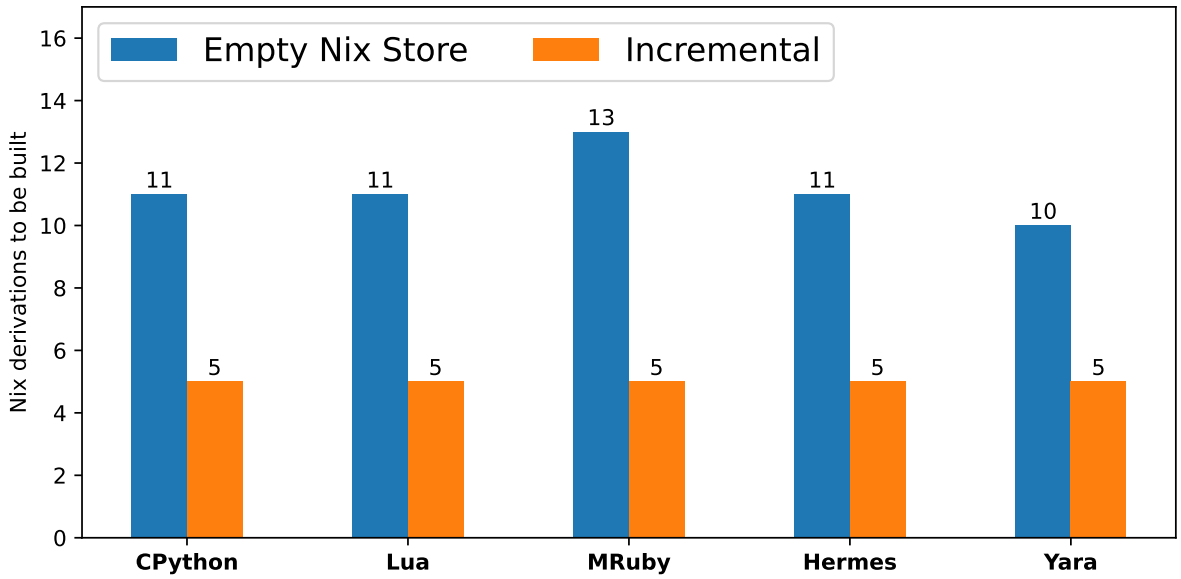

Figure A.4 – *Results of the same experiment presented in Figure A.3, but this time we display the number of Nix derivations that had to be built. The reduction of the number of required derivations is greater than in Figure A.2 as we do not need to rebuild the fuzzer (Grammar-Mutator).*

Yara Module	vanilla AFL++	Grammar-Mutator
lexer.l	0.56	0.45
lexer.c	0.48	0.31
re_lexer.l	0.68	0.04
re_lexer.c	0.23	0.15
hex_lexer.l	0.51	0.27
hex_lexer.c	0.17	0.11
hex_grammar.y	1	0.81
hex_grammar.c	0.51	0.40
re_grammar.y	0.95	0.21
re_grammar.c	0.42	0.26
...	...	...
object.c	0.20	0.38
string.c	0.05	0.16
exec.c	0.17	0.25
scan.c	0.36	0.40

Table A.1 – *Obtained edge coverage (from 0 to 1) in some modules of Yara when fuzzing the target with vanilla AFL++ and Grammar-Mutator. The upper group of modules is related to the frontend of the target. Here vanilla AFL++ naturally achieves better results. On the other hand, the lower group of modules in the table actually implement real, "deep" logic of the Yara language. Grammar-Mutator outperformed vanilla AFL++ in this case. Naturally, modules in the lower group contain more edges in absolute terms.*