# GRIMOIRE: Synthesizing Structure while Fuzzing

Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi,
Sergej Schumilo, Simon Wörner and Thorsten Holz

*Ruhr-Universität Bochum, Germany*

## Abstract

In the past few years, fuzzing has received significant attention from the research community. However, most of this attention was directed towards programs without a dedicated parsing stage. In such cases, fuzzers which leverage the input structure of a program can achieve a significantly higher code coverage compared to traditional fuzzing approaches. This advancement in coverage is achieved by applying large-scale mutations in the application's input space. However, this improvement comes at the cost of requiring expert domain knowledge, as these fuzzers depend on structure input specifications (e. g., grammars). Grammar inference, a technique which can automatically generate such grammars for a given program, can be used to address this shortcoming. Such techniques usually infer a program's grammar in a pre-processing step and can miss important structures that are uncovered only later during normal fuzzing.

In this paper, we present the design and implementation of GRIMOIRE, a fully automated coverage-guided fuzzer which works without any form of human interaction or pre-configuration; yet, it is still able to efficiently test programs that expect highly structured inputs. We achieve this by performing large-scale mutations in the program input space using grammar-like combinations to synthesize new highly structured inputs without any pre-processing step. Our evaluation shows that GRIMOIRE outperforms other coverage-guided fuzzers when fuzzing programs with highly structured inputs. Furthermore, it improves upon existing grammar-based coverage-guided fuzzers. Using GRIMOIRE, we identified 19 distinct memory corruption bugs in real-world programs and obtained 11 new CVEs.

## 1 Introduction

As the amount of software impacting the (digital) life of nearly every citizen grows, effective and efficient testing mechanisms for software become increasingly important. The publication of the fuzzing framework AFL [65] and its success at uncovering a huge number of bugs in highly relevant software has spawned a large body of research on effective feedback-based fuzzing. AFL and its derivatives have largely conquered automated, dynamic software testing and are used to uncover new security issues and bugs every day. However, while great progress has been achieved in the field of fuzzing, many hard cases still require manual user interaction to generate satisfying test coverage. To make fuzzing available to more programmers and thus scale it to more and more target programs, the amount of expert knowledge that is required to effectively fuzz should be reduced to a minimum. Therefore, it is an important goal for fuzzing research to develop fuzzing techniques that require less user interaction and, in particular, less domain knowledge to enable more automated software testing.

**Structured Input Languages.** One common challenge for current fuzzing techniques are programs which process highly structured input languages such as interpreters, compilers, text-based network protocols or markup languages. Typically, such inputs are consumed by the program in two stages: parsing and semantic analysis. If parsing of the input fails, deeper parts of the target program—containing the actual application logic—fail to execute; hence, bugs hidden "deep" in the code cannot be reached. Even advanced feedback fuzzers—such as AFL—are typically unable to produce diverse sets of syntactically valid inputs. This leads to an imbalance, as these programs are part of the most relevant attack surface in practice, yet are currently unable to be fuzzed effectively. A prominent example are browsers, as they parse a multitude of highly-structured inputs, ranging from XML or CSS to JavaScript and SQL queries.

Previous approaches to address this problem are typically based on manually provided grammars or seed corpora [2, 14, 45, 52]. On the downside, such methods require human experts to (often manually) specify the grammar or suitable seed corpora, which becomes next to impossible for applications with undocumented or proprietary input specifications. An orthogonal line of work tries to utilize advanced program analysis techniques to automatically infer grammars

[4, 5, 25]. Typically performed as a pre-processing step, such methods are used for generating a grammar that guides the fuzzing process. However, since this grammar is treated as immutable, no additional learning takes place during the actual fuzzing run.

**Our Approach.** In this paper, we present a novel, fully automated method to fuzz programs with a highly structured input language, without the need for any human expert or domain knowledge. Our approach is based on two key observations: First, we can use code coverage feedback to automatically infer structural properties of the input language. Second, the precise and "correct" grammars generated by previous approaches are actually unnecessary in practice: since fuzzers have the virtue of high test case throughput, they can deal with a significant amount of noise and imprecision. In fact, in some programs (such as `Boolector`) with a rather diverse set of input languages, the additional noise even benefits the fuzz testing. In a similar vein, there are often program paths which can only be accessed by inputs *outside* of the formal specifications, e. g., due to incomplete or imprecise implementations or error handling code.

Instead of using a pre-processing step, our technique is directly integrated in the fuzzing process itself. We propose a set of generalizations and mutations that resemble the inner workings of a grammar-based fuzzer, without the need for an explicit grammar. Our generalization algorithm analyzes each newly found input and tries to identify substrings of the input which can be replaced or reused in other positions. Based on this information, the mutation operators recombine fragments from existing inputs. Overall, this results in synthesizing new, structured inputs without prior knowledge of the underlying specification.

We have implemented a prototype of the proposed approach in a tool called GRIMOIRE[1]. GRIMOIRE does not need any specification of the input language and operates in an automated manner without requiring human assistance; in particular, without the need for a format specification or seed corpus. Since our techniques make no assumption about the program or its environment behavior, GRIMOIRE can be easily applied to closed-source targets as well.

To demonstrate the practical feasibility of our approach, we perform a series of experiments. In a first step, we select a diverse set of programs for a comparative evaluation: we evaluate GRIMOIRE against other fuzzers on four scripting language interpreters (`mruby`, `PHP`, `Lua` and `JavaScriptCore`), a compiler (`TCC`), an assembler (`NASM`), a database (`SQLite`), a parser (`libxml`) and an SMT solver (`Boolector`). Demonstrating that our approach can be applied in many different scenarios without requiring any kind of expert knowledge, such as an input specification. The evaluation results show that our approach outperforms all existing coverage-guided fuzzers; in the case of `Boolector`, GRIMOIRE finds up to 87% more coverage than the baseline (REDQUEEN). Second, we evaluate GRIMOIRE against state-of-the-art grammar-based fuzzers. We observe that in situations where an input specification is available, it is advisable to use GRIMOIRE in addition to a grammar fuzzer to further increase the test coverage found by grammar fuzzers. Third, we evaluate GRIMOIRE against current state-of-the-art approaches that use automatically inferred grammars for fuzzing and found that we can significantly outperform such approaches. Overall, GRIMOIRE found 19 distinct memory corruption bugs that we manually verified. We responsibly disclosed all of them to the vendors and obtained 11 CVEs. During our evaluation, the next best fuzzer only found 5 of these bugs. In fact, GRIMOIRE found more bugs than all five other fuzzers combined.

[1] A *grimoire* is a magical book that recombines magical elements to formulas. Furthermore, it has the same word stem as the Old French word for grammar—namely, *gramaire*.

**Contributions.** In summary, we make the following contributions:

- We present the design, implementation and evaluation of GRIMOIRE, an approach to fully automatically fuzz highly structured formats with no human interaction.
- We show that even though GRIMOIRE is a binary-only fuzzer that needs no seeds or grammar as input, it still outperforms many fuzzers that make significantly stronger assumptions (e. g., access to seeds, grammar specifications and source code).
- We found and reported multiple bugs in various common projects such as `PHP`, `gnuplot` and `NASM`.

## 2 Challenges in Fuzzing Structured Languages

In this section, we briefly summarize essential information paramount to the understanding of our approach. To this end, we provide an overview of different fuzzing approaches, while focusing on their shortcomings and open challenges. In particular, we describe those details of AFL (e. g., code coverage) that are necessary to understand our approach. Additionally, we explain how fuzzers explore the state space of a program and how grammars aid the fuzzing process.

Generally speaking, fuzzing is a popular and efficient software testing technique used to uncover bugs in applications. Fuzzers typically operate by producing a large number of test cases, some of which may trigger bugs. By closely monitoring the runtime execution of these test cases, fuzzers are able to locate inputs causing faulty behavior. In an abstract view, one can consider fuzzing as randomly exploring the state space of the application. Typically, most totally random inputs are rejected early by the target application and do not visit interesting parts of the state space. Thus, in our abstract view, the state space has interesting and uninteresting regions. Efficient fuzzers somehow have to ensure that they avoid uninteresting regions most of the time. Based on this observation, we can divide fuzzers into three broad categories, namely: (a) blind, (b) coverage-guided and (c) hybrid fuzzers, as explained next.

## 2.1 Blind Fuzzing

The most simple form of a fuzzer is a program which generates a stream of random inputs and feeds it to the target application. If the fuzzer generates inputs without considering the internal behavior of the target application, it is typically referred to as a *blind fuzzer*. Examples of blind fuzzers are RADAMSA [29], PEACH [14], Sulley [45] and ZZUF [32]. To obtain new inputs, fuzzers traditionally can build on two strategies: *generation* and *mutation*.

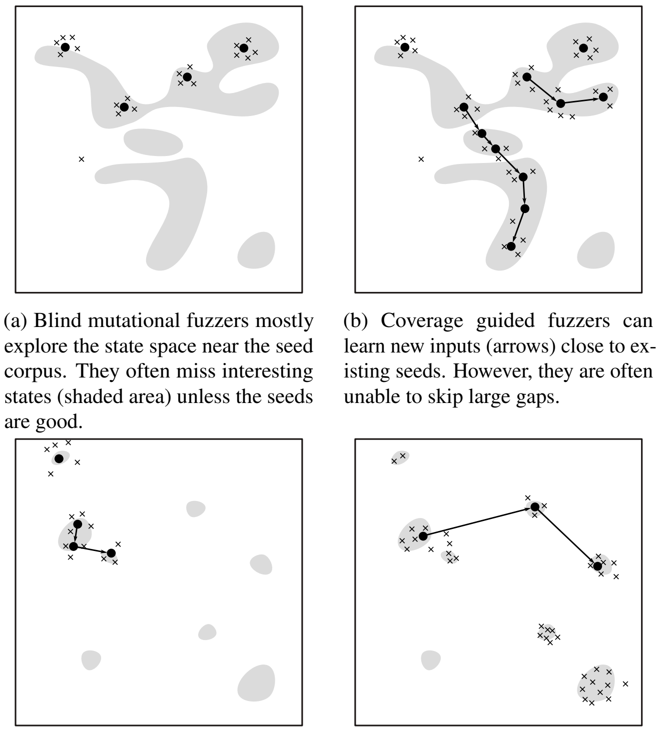
Fuzzers employing the former approach have to acquire a specification, typically a grammar or model, of an application's expected input format. Then, a fuzzer can use the format specification to be able to generate novel inputs in a somewhat efficient way. Additionally, in some cases, a set of valid inputs (a so-called *corpus*) might be required to aid the generation process [46, 58].

On the other hand, fuzzers which employ a mutation-based strategy require only an initial corpus of inputs, typically referred to as *seeds*. Further test cases are generated by randomly applying various mutations on initial seeds or novel test cases found during fuzzing runs. Examples for common mutators include bit flipping, splicing (i. e., recombining two inputs) and repetitions [14, 29, 32]. We call these mutations *small-scale mutations*, as they typically change small parts of the program input.

Blind fuzzers suffer from one major drawback. They either require an extensive corpus or a well-designed specification of the input language to provide meaningful results. If a program feature is not represented by either a seed or the input language specification, a blind fuzzer is unlikely to exercise it. In our abstract, state space-based view, this can be understood as blindly searching the state space near the seed inputs, while failing to explore interesting neighborhoods, as illustrated in Figure 1(a). To address this limitation, the concept of coverage-guided fuzzing was introduced.

## 2.2 Coverage-guided Fuzzing

Coverage-guided fuzzers employ lightweight program coverage measurements to trace how the execution path of the application changes based on the provided input (e. g., by tracking which basic blocks have been visited). These fuzzers use this information to decide which input should be stored or discarded to extend the corpus. Therefore, they are able to evolve inputs that differ significantly from the original seed corpus



(a) Blind mutational fuzzers mostly explore the state space near the seed corpus. They often miss interesting states (shaded area) unless the seeds are good.

(b) Coverage guided fuzzers can learn new inputs (arrows) close to existing seeds. However, they are often unable to skip large gaps.

(c) Programs with highly structured input formats typically have large gaps in the state space. Current feedback and hybrid fuzzers have difficulties finding other interesting islands using local mutations.

(d) By introducing an input specification, fuzzers can generate inputs in interesting areas and perform large-scale mutations that allow to jump between islands of interesting states.

Figure 1: Different fuzzers exploring distinct areas in state space.

while at the same time exercising new program features. This strategy allows to gradually explore the state of the program as it uncovers new paths. This behavior is illustrated in Figure 1(b). The most prominent example of a coverage-guided fuzzer is AFL [65]. Following the overwhelming success of AFL, various more efficient coverage-guided fuzzers such as ANGORA [12], QSYM [64], T-FUZZ [47] or REDQUEEN [3] were proposed.

From a high-level point of view, all these AFL-style fuzzers can be broken down into three different components: (i) the input queue stores and schedules all inputs found so far, (ii) the mutation operations produce new variants of scheduled inputs and (iii) the global coverage map is used to determine whether a new variant produced novel coverage (and thus should be stored in the queue).

From a technical point of view, this maps to AFL as follows: Initially, AFL fills the input queue with the seed inputs. Then, it runs in a continuous fuzzing loop, composed of the following steps: (1) Pick an input from the input queue, then (2) apply multiple mutation operations on it. After each mutation, (3) execute the target application with the selected input. If new coverage was triggered by the input, (4) save it back to the queue. To determine whether new coverage was triggered, AFL compares the results of the execution with the values in the global coverage map.

This global coverage map is filled as follows: AFL shares a memory area of the same size as the global coverage map with the fuzzing target. During execution, each transition between two basic blocks is assigned a position inside this shared memory. Every time the transition is triggered, the corresponding entry (one byte) in the shared memory map is incremented. To reduce overhead incurred by large program traces, the shared coverage map has a fixed size (typically $2^{16}$ bytes). While this might introduce collisions, empirical evaluation has shown that the performance gains make up for the loss in the precision [66].

After the target program terminates, AFL compares the values in the shared map to all previous runs stored in the global coverage map. To check if a new edge was executed, AFL applies the so-called *bucketing*. During bucketing, each entry in the shared map is rounded to a power of 2 (i. e., at most a single bit is set in each entry). Then, a simple binary operation is used to check if any new bits are present in the shared map (but not the global map). If any new bit is present, the input is stored in the queue. Furthermore, all new bits are also set to 1 in the global coverage map. We distinguish between *new bits* and *new bytes*. If a new bit is set to 1 in a byte that was previously zero, we refer to it as a *new byte*. Intuitively, a new byte corresponds to new coverage while a new bit only illustrates that a known edge was triggered more often (e. g., more loop iterations were observed).

**Example 1.** *For example, consider some execution a while after starting the fuzzer run for a program represented by its Control-Flow Graph (CFG) in Figure 2ⓐ. Assume that the fictive execution of an input causes a loop between B and C to be executed 10 times. Hence, the shared map is updated as shown in ⓑ, reflecting the fact that edges $A \rightarrow B$ and $C \rightarrow D$ were executed only once, while the edges $B \rightarrow C$ and $C \rightarrow B$ were encountered 10 (*`0b1010`*) times. In ⓒ, we illustrate the final bucketing step. Note how* `0b1010` *is put into the bucket* `0b1000`*, while* `0b0001` *is moved into the one identified by* `0b0001`*. Finally, AFL checks whether the values encountered in this run triggered unseen edges in ⓓ. To this end, we compare the shared map to the global coverage map and update it accordingly (see ⓔ), setting bits set in the shared but not global coverage map. As visualized in ⓕ, a new bit was set for two entries, while a new byte was found for one. This means that the edge between $C \rightarrow D$ was previously unseen, thus the input used for this example triggered new coverage.*

While coverage-guided fuzzers significantly improve upon blind fuzzers, they can only learn from new coverage if they are able to guess an input that triggers the new path in the program. In certain cases, such as multi-byte magic values, the probability of guessing an input necessary to trigger a different path is highly unlikely. These kind of situations occur if there is a significant gap between interesting areas in the state space and existing mutations are unlikely to cross the uninteresting gap. The program displayed in the Figure 1(b) illustrates a case with only one large gap in the program space. Thus, this program is well-suited for coverage-guided fuzzing. However, current mutation-based coverage-guided fuzzers struggle to explore the whole state space because the island in the lower right is never reached. To overcome this limitation, hybrid fuzzer were introduced; these combine coverage-guided fuzzing with more in-depth program analysis techniques.

## 2.3 Hybrid Fuzzing

Hybrid fuzzers typically combine coverage-guided fuzzing with program analysis techniques such as symbolic execution, concolic execution or taint tracking. As noted above, fast and cheap fuzzing techniques can uncover the bulk of the easy-to-reach code. However, they struggle to trigger program paths that are highly unlikely. On the other hand, symbolic or concolic execution does not move through the state space randomly. Instead, these techniques use an SMT solver to find inputs that trigger the desired behavior. Therefore, they can cover hard-to-reach program locations. Still, as a consequence of the precise search technique, they struggle to explore large code regions due to significant overhead.

By combining fuzzing and reasoning-based techniques, one can benefit from the strength of each individual technique, while avoiding the drawbacks. Purely symbolic approaches have proven difficult to scale. Therefore, most current tools such as Sage [21], Driller [54] or QSym [64] use concolic execution instead. This mostly avoids the state explosion problem by limiting the symbolic execution to a single path. To further reduce the computation cost, some fuzzers such as VUzzer [50] and Angora [12] only use taint tracking. Both approaches still allow to overcome the common multi-byte magic value problem. However, they lose the ability to explore behavior more globally.

While hybrid fuzzers can solve constraints over individual values of the input, they are typically not efficient at solving constraints on the *overall* structure of the input. Consider target programs such as a script interpreter. To uncover a new valid code path, the symbolic executor usually has to consider a completely different path through the parsing stage. This leads to a large number of very large gaps in the state space as illustrated in Figure 1(c). Therefore, concolic execution or taint tracking-based tools are unable to solve these constraints. In purely symbolic execution-based approaches, this leads to a massive state explosion.

## 2.4 Coverage-guided Grammar Fuzzing

Beside the problem of multi-byte magic values, there is another issue which leads to large gaps between interesting

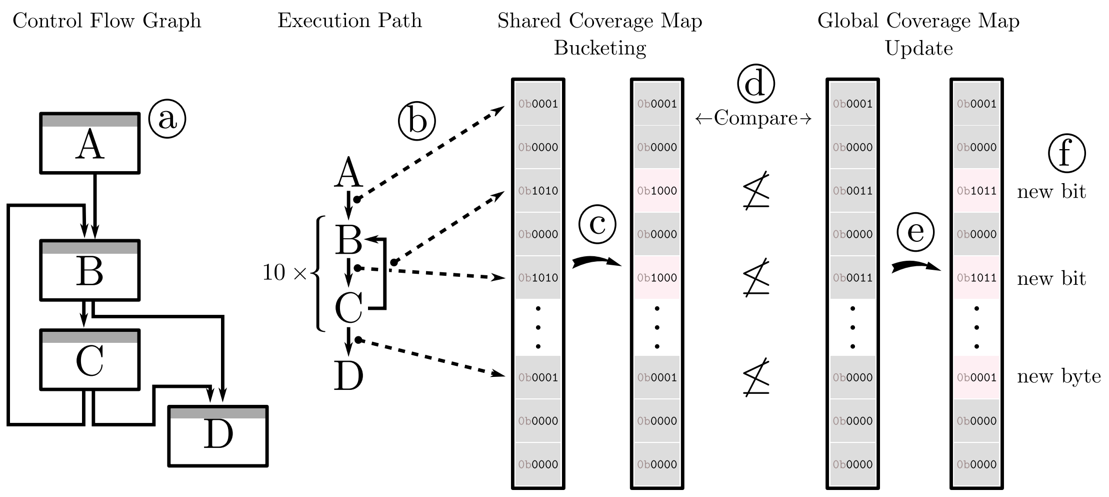


Figure 2: The process of tracing a path in a program and introducing new bits and bytes in the global coverage map.

parts of the state space: programs with structured input languages. Examples for such programs are interpreters, compilers, databases and text-based Internet protocols. As mentioned earlier, current mutational blind and coverage-guided as well as hybrid fuzzers cannot efficiently fuzz programs with structured input languages. To overcome this issue, *generational fuzzers* (whether blind, coverage-guided or hybrid) use a specification of the input language (often referred to as a *grammar*) to generate valid inputs. Thereby, they reduce the space of possible inputs to a subset that is much more likely to trigger interesting states. Additionally, coverage-guided grammar fuzzers can mutate inputs in this reduced subset by using the provided grammar. We call these mutations *large-scale mutations* since they modify large part of the input. This behavior is illustrated in Figure 1(d).

Therefore, the performance of fuzzers can be increased drastically by providing format specifications to the fuzzer, as implemented in NAUTILUS [2] and AFLSMART [48]. These specifications let the fuzzer spend more time exercising code paths deep in the target application. Particularly, the fuzzer is able to sensibly recombine inputs that trigger interesting features in a way that has a good chance of triggering more interesting behaviors.

Grammar fuzzers suffer from two major drawbacks. First, they require human effort to provide precise format specification. Second, if the specification is incomplete or inaccurate, the fuzzer lacks the capability to address these shortcomings. One can overcome these two drawbacks by automatically inferring the specification (grammar).

## 2.5 Grammar Inference

Due to the impact of grammars on software testing, various approaches have been developed that automatically can generate input grammars for target programs. Bastani et al. [5] introduced GLADE, which uses a modified version of the target as a black-box oracle that tests if a given input is syntactically valid. GLADE turns valid inputs into regular expressions that generate (mostly) valid inputs. Then, these regular expressions are turned into full grammars by trying to introduce recursive replacement rules. In each step, the validity of the resulting grammar is tested using multiple oracle queries. This approach has three significant drawbacks: First, the inference process takes multiple hours for complex targets such as scripting languages. Second, the user needs to provide an automated testing oracle, which might not be trivial to produce. Third, in the context of fuzzing, the resulting grammars are not well suited for fuzzing as our evaluation shows (see Section 5.4 for details). Additionally, this approach requires a pre-processing step before fuzzing starts in order to infer a grammar from the input corpus.

Other approaches use the target application directly and thus avoid the need to create an oracle. AUTOGRAM [34], for instance, uses the original program and taint tracking to infer grammars. It assumes that the functions that are called during parsing reflect the non-terminals of the intended grammar. Therefore, it does not work for recursive descent parsers. PYGMALION [25] is based on simplified symbolic execution of Python code to avoid the dependency on a set of good inputs. Similar to AUTOGRAM, PYGMALION assumes that the function call stack contains relevant information to identify recursive rules in the grammar. This approach works well for hand-written, recursive descent parsers; however, it will have severe difficulties with parsers generated by parser generators. These parsers are typically implemented as table-driven automatons and do not use function calls at all. Additionally, robust symbolic execution and taint tracking are still challenging for binary-only targets.

## 2.6 Shortcomings of Existing Approaches

To summarize, current automated software testing approaches have the following disadvantages when used for fuzzing of programs that accept structured input languages:

- **Needs Human Assistance.** Some techniques require human assistance to function properly. Either in terms of providing information or in terms of modifying the target program.
- **Requires Source Code.** Some fuzzing techniques require access to source code. This puts them at a disadvantage as they cannot be applied to proprietary software in binary format.
- **Requires a Precise Environment Model.** Techniques based on formal reasoning such as symbolic/concolic execution as well as taint tracking require precise semantics of the underlying platform as well as semantics of all used Operating System (OS) features (e. g., syscalls).
- **Requires a Good Corpus.** Many techniques only work if the seed corpus already contains most features of the input language.
- **Requires a Format Specification.** Similarly, many techniques described in this section require precise format specifications for structured input languages.
- **Limited To Certain Types of Parsers.** Some approaches make strong assumptions about the underlying implementation of the parser. Notably, some approaches are unable to deal with parses generated by common parser generators such as `GNU Bison` [15] or `Yacc` [37].
- **Provides Only Small-scale Mutations.** As discussed in this section, various approaches cannot provide mutations that cross large gaps in the program space.

Table 1: Requirements and limitations of different fuzzers and inference tools when used for fuzzing structured input languages. If a shortcoming applies to a tool, it is denoted with ✗, otherwise with ✓.

| | PEACH | AFL | REDQUEEN | QSYM | ANGORA | NAUTILUS | AFLSMART | GLADE | AUTOGRAM | PYGMALION | GRIMOIRE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| human assistance | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| source code | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ |
| environment model | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| good corpus | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| format specifications | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| certain parsers | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| small-scale mutations | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

We analyzed existing fuzzing methods, the results of this survey are shown in Table 1. We found that all current approaches have at least one shortcoming for fuzzing programs with highly structured inputs. In the next section, we propose a design that avoids all the mentioned drawbacks.

## 3 Design

Based on the challenges identified above, we now introduce the design of GRIMOIRE, a fully automated approach that synthesizes the target's structured input language during fuzzing. Furthermore, we present large-scale mutations that cross significant gaps in the program space. Note that none of the limitations listed in Table 1 applies to our approach. To emphasize, our design does not require any previous information about the input structure. Instead, we learn an ad-hoc specification based on the program semantics and use it for coverage-guided fuzzing.

We first provide a high-level overview of GRIMOIRE, followed by a detailed description. GRIMOIRE is based on identifying and recombining fragments in inputs that trigger new code coverage during a normal fuzzing session. It is implemented as an additional fuzzing stage on top of a coverage-guided fuzzer. In this stage, we strip every new input (that is found by the fuzzer and produced new coverage) by replacing those parts of the input that can be modified or replaced without affecting the input's new coverage by the symbol □. This can be understood as a generalization, in which we reduce inputs to the fragments that trigger new coverage, while maintaining information about *gaps* or *candidate positions* (denoted by □). These gaps are later used to splice in fragments from other inputs.

**Example 2.** *Consider the input "*`if(x>1) then x=3 end`*" and assume it was the first input to trigger the coverage for a syntactically correct if-statement as well as for "*`x>1`*". We can delete the substring "*`x=3`*" without affecting the interesting new coverage since the if-statement remains syntactically correct. Additionally, the space between the condition and the "*`then`*" is not mandatory. Therefore, we obtain the generalized input "*`if(x>1)□then □end`*".*

After a set of inputs was successfully generalized, fragments from the generalized inputs are recombined to produce new candidate inputs. We incorporate various different strategies to combine existing fragments, learned tokens (a special form of substrings) and strings from the binary in an automated manner.

**Example 3.** *Assume we obtained the following generalized inputs: "*`if(x>1)□then □end`*" and "*`□x=□y+□`*". We can use this information in many ways to generate plausible recombinations. For example, starting with the input "*`if(x>1)□then □end`*", we can replace the second gap with the second input, obtaining "*`if(x>1)□then □x=□y+□end`*". Afterwards, we choose the slice "*`□y+□`*" from the second input and splice it into the fourth gap and obtain "*`if(x>1)□then □x=□y+□y+□end`*". In a last step,* *we replace all remaining gaps by an empty string. Thus, the final input is* “`if(x>1)then x=y+y+end`”.

One could think of our approach as a context-free grammar with a single non-terminal input □ and all fragments of generalized inputs as production rules. Using these loose, grammar-like recombination methods in combination with feedback-driven fuzzing, we are able to automatically learn interesting structures.

## 3.1 Input Generalization

We try to generalize inputs that produced new coverage (e. g., inputs that introduced new bytes to the bitmap, cf. Section 2.2). The generalization process (Algorithm 1) tries to identify parts of the input that are irrelevant and fragments that caused new coverage. In a first step, we use a set of rules to obtain fragment boundaries (Line 3). Consecutively, we remove individual fragments (Line 4). After each step, we check if the reduced input still triggers the same new coverage bytes as the original input (Line 5). If this is the case, we replace the fragment that was removed by a □ and keep the reduced input (Line 6).

**Algorithm 1:** Generalizing an input through fragment identification.

```
Data: input is the input to generalize, new_bytes are the new
      bytes of the input, splitting_rule defines how to split an
      input
Result: A generalized version of input
1 start ← 0
2 while start < input.length() do
3     end ← find_next_boundary(input, splitting_rule)
4     candidate ← remove_substring(input, start, end)
5     if get_new_bytes(candidate) == new_bytes then
6         input ← replace_by_gap(input, start, end)
7     start ← end
8 input ← merge_adjacent_gaps(input)
```

**Example 4.** *Consider input* “`pprint 'aaaa'`” *triggers the new bytes* 20 *and* 33 *because of the pprint statement. Furthermore, assume that we use a rule that splits inputs into non-overlapping chunks of length two. Then, we obtain the chunks* “`pp`”, “`ri`”, “`nt`”, “` '`”, “`aa`”, “`aa`” *and* “`'`”. *If we remove any of the first four chunks, the modified input will not trigger the same new bytes since we corrupted the pprint statement. However, if we remove the fifth or sixth chunk, we still trigger the bytes* 20 *and* 33 *since the pprint statement remains valid. Therefore, we reduce the input to* “`pprint '□□'`”. *As we have two adjacent* □*, we merge them into one. The generalized input is* “`pprint '□'`”.

To generalize an input as much as possible, we use several fragmentation strategies for which we apply Algorithm 1 repeatedly. First, we split the input into overlapping chunks of size 256, 128, 64, 32, 2 and 1 to remove large uninteresting parts as early as possible. Afterwards, we dissect at different separators such as ‘`.`’, ‘`;`’, ‘`,`’, ‘`\n`’, ‘`\r`’, ‘`\t`’, ‘`#`’ and ‘` `’. As a consequence, we can remove one or more statements in code, comments and other parts that did not cause the input’s new coverage. Finally, we split at different kinds of brackets and quotation marks. These fragments can help to generalize constructs such as function parameters or nested expressions. In detail, we split in between of ‘`()`’, ‘`[]`’, ‘`{}`’, ‘`<>`’ as well as single and double quotes. To guess different nesting levels in between these pairs of opening/closing characters, we extend Algorithm 1 as follows: If the current index `start` matches an opening character, we search the furthermost matching closing character, create a `candidate` by removing the substring in between and check if it triggers the same new coverage. We iteratively do this by choosing the next furthermost closing character—effectively shrinking the fragment size—until we find a substring that can be removed without changing the `new_bytes` or until we reach the index `start`. In doing so, we are able to remove the largest matching fragments from the input that are irrelevant for the input’s new coverage.

Since we want to recombine (generalized) inputs to find new coverage—as we describe in the following section—we store the original input as well as its generalization. Furthermore, we split the generalized input at every □ and store the substrings *(tokens)* in a set; these tokens often are syntactically interesting fragments of the structured input language.

**Example 5.** *We map the input* “`if(x>1) then x=3 end`” *to its generalization* “`if(x>1)□then □end`”. *In addition, we extract the tokens* “`if(x>1)`”, “`then `” *and* “`end`”. *For the generalized input* “`□x=□y+□`”, *we remember the tokens* “`x=`” *and* “`y+`”.

## 3.2 Input Mutation

GRIMOIRE builds upon knowledge obtained from the generalization stage to generate inputs that have good chances of finding new coverage. For this, it recombines (fragments of) generalized inputs, tokens and strings (stored in a dictionary) that are automatically obtained from the data section of the target’s binary. On a high level, we can divide our mutations into three standalone operations: input extension, recursive replacement and string replacement.

Given the current input from the fuzzing queue, we add these mutations to the so-called *havoc phase* [3] as described in Algorithm 2. First, we use Redqueen’s `havoc_amount` to determine—based on the input’s performance—how often we should apply the following mutations (in general, between 512 and 1024 times). First, if the input triggered new bytes in the bitmap, we take its generalized form and apply the large-scale mutations `input_extension` and `recursive_replacement`. Afterwards, we take the original input string (accessed by `input.content()`) and apply the

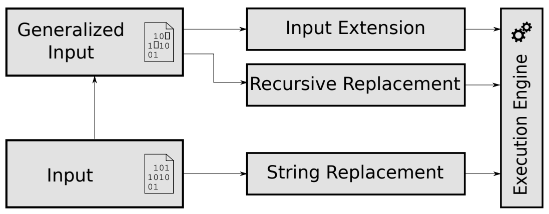


Figure 3: A high-level overview of our mutations. Given an input, we apply various mutations on its generalized and original form. Each mutation then feeds mutated variants of the input to the fuzzer's execution engine.

`string_replacement` mutation. This process is illustrated in Figure 3.

**Algorithm 2:** High-level overview of the mutations introduced in GRIMOIRE.

```
Data: input is the current input in the queue, generalized is the
      set of all previously generalized inputs, tokens and strings
      from the dictionary, strings is the provided dictionary
      obtained from the binary
1 content ← input.content()
2 n ← havoc_amount(input.performance())
3 for i ← 0 to n do
4     if input.is_generalized() then
5         input_extension(input, generalized)
6         recursive_replacement(input, generalized)
7     string_replacement(content, strings)
```

Before we describe our mutations in detail, we explain two functions that all mutations have in common—`random_generalized` and `send_to_fuzzer`. The function `random_generalized` takes as input a set of all previously generalized inputs, tokens and strings from the dictionary and returns—based on random coin flips—a random (slice of a ) generalized input, token or string. In case we pick an input slice, we select a substring between two arbitrary □ in a generalized input. This is illustrated in Algorithm 3. The other function, `send_to_fuzzer`, implies that the fuzzer executes the target application with the mutated input. It expects concrete inputs. Thus, mutations working on generalized inputs first replace all remaining □ by an empty string.

**Algorithm 3:** Random selection of a generalized input, slice, token or string.

```
Data: generalized is the set of all previously generalized inputs,
      tokens and strings from the dictionary
Result: rand is a random generalized input, slice token or string
1 if random_coin() then
2     if random_coin() then
3         rand ← random_slice(generalized)
4     else
5         rand ← random_token_or_string(generalized)
6 else
7     rand ← random_generalized_input(generalized)
```

### 3.2.1 Input Extension

The input extension mutation is inspired by the observation that—in highly structured input languages—often inputs are chains of syntactically well-formed statements. Therefore, we extend an generalized input by placing another randomly chosen generalized input, slice, token or string before and after the given one. This is described in Algorithm 4.

**Algorithm 4:** Overview of the input extension mutation.

```
Data: input is the current generalized input, generalized is the
      set of all previously generalized inputs, tokens and strings
      from the dictionary
1 rand ← random_generalized(generalized_inputs)
2 send_to_fuzzer(concat(input.content(),
  rand.content()))
3 send_to_fuzzer(concat(rand.content(),
  input.content()))
```

**Example 6.** *Assume that the current input is "*`pprint 'aaaa'`*" and its generalization is "*`pprint '□'`*". Furthermore, assume that we randomly choose a previous generalization "*`□x=□y+□`*". Then, we concretize their generalizations to "*`pprint '$$'`*" and "*`x=y+`*" by replacing remaining gaps with an empty string. Afterwards, we concatenate them and obtain "*`pprint '$$'x=y+`*" and "*`x=y+pprint '$$'`*".*

### 3.2.2 Recursive Replacement

The recursive replacement mutation recombines knowledge about the structured input language—that was obtained earlier in the fuzzing run—in a grammar-like manner. As illustrated in Algorithm 5, given a generalized input, we extend its beginning and end by □—if not yet present—such that we always can place other data before or behind the input. Afterwards, we randomly select $n \in \{2,4,8,16,32,64\}$ and perform the following operations $n$ times: First, we randomly select another generalized input, input slice, token or string. Then, we call `replace_random_gap` which replaces an arbitrary □ in the first generalized input by the chosen element. Furthermore, we enforce □ before and after the replacement such that these □ can be subject to further replacements. Finally, we concretize the mutated input and send it to the fuzzer. The recursive replacement mutator has a (comparatively) high likelihood of producing new structurally interesting inputs compared to more small-scale mutations used by current coverage-guided fuzzers.

**Example 7.** *Assume that the current input is "*`pprint 'aaaa'`*". We take its generalization "*`pprint '□'`*" and extend it to "*`□pprint '□'□`*". Furthermore, assume that we already generalized the inputs "*`if(x>1)□then □end`*" and "*`□x=□y+□`*". In a first mutation, we choose to replace the first □ with the slice "*`if(x>1)□`*". We extend the slice to "*`□if(x>1)□`*" and obtain "*`□if(x>1)□pprint`

**Algorithm 5:** Overview of the recursive replacement mutation.

```
Data: input is the current generalized input, generalized is the
      set of all previously generalized inputs, tokens and strings
      from the dictionary
input ← pad_with_gaps(input)
for i ← 0 to random_power_of_two() do
    rand ← random_generalized(generalized_inputs)
    input ← replace_random_gap(input, rand)
send_to_fuzzer(input.content())
```

'□'□". *Afterwards, we choose to replace the third □ with* "□x=□y+□" *and obtain* "□if(x>1)□pprint '□x=□y+□'□". *In a final step, we replace the remaining □ with an empty string and obtain* "if(x>1)pprint 'x=y+'".

### 3.2.3 String Replacement

Keywords are important elements of structured input languages; changing a single keyword in an input can lead to completely different behavior. GRIMOIRE's string replacement mutation performs different forms of replacements, as described in Algorithm 6. Given an input, it locates all substrings in the input that match strings from the obtained dictionary and chooses one randomly. GRIMOIRE first selects a random occurrence of the matching substring and replaces it with a random string. In a second step, it replaces all occurrences of the substring with the same random string. Finally, the mutation sends both mutated inputs to the fuzzer. As an example, this mutation can be helpful to discover different methods of the same object by replacing a valid method call with different alternatives. Also, changing all occurrences of a substring allows us to perform more syntactically correct mutations, such as renaming of variables in the input.

**Example 8.** *Assume the* "if(x>1)pprint 'x=y+'" *and that the strings "if", "while", "key", "pprint", "eval", "+", "=" and "–" are in the dictionary. Thus, the string replacement mutation can generate inputs such as* "while(x>1)pprint 'x=y+'", "if(x>1)eval 'x+y+'" *or* "if(x>1)pprint 'x=y-'". *Furthermore, assume that the string "x" is also in the dictionary. Then, the string replacement mutation can replace all occurrences of the variable* "x" *in* "if(x>1)pprint 'x=y+'" *and obtain* "if(key>1)pprint 'key=y+'".

## 4 Implementation

To evaluate the algorithms introduced in this paper, we built a prototype implementation of our design. Our implementation, called GRIMOIRE, is based on REDQUEEN's [3] source code. This allows us to implement our techniques within a state-of-the-art fuzzing framework. REDQUEEN is applicable to both open and closed source targets running in user or kernel space, thus enabling us to target a wide variety of programs.

**Algorithm 6:** Overview of the string replacement mutation.

```
Data: input is the input string, strings is the provided dictionary
      obtained from the binary
sub ← find_random_substring(input, strings)
if sub then
    rand ← random_string(strings)
    data ← replace_random_instance(input, sub, rand)
    send_to_fuzzer(data)
    data ← replace_all_instances(input, sub, and)
    send_to_fuzzer(data)
```

While REDQUEEN is entirely focused on solving magic bytes and similar constructs which are local in nature (i. e., require only few bytes to change), GRIMOIRE assumes that this kind of constraints can be solved by the underlying fuzzer. It uses global mutations (that change large parts of the input) based on the examples that the underlying fuzzer finds. Since our technique is merely based on common techniques implemented in coverage-guided fuzzers—for instance, access to the execution bitmap—it would be a feasible engineering task to adapt our approach to other current fuzzers, such as AFL.

More precisely, GRIMOIRE is implemented as a set of patches to REDQUEEN. After finding new inputs, we apply the generalization instead of the minimization algorithm that was used by AFL and REDQUEEN. Additionally, we extended the havoc stage by large-scale mutations as explained in Section 3. To prevent GRIMOIRE from spending too much time in the generalization phase, we set a user-configurable upper bound; inputs whose length exceeds this bound are not be generalized. Per default, it is set to 16384 bytes. Overall, about 500 lines were written to implement the proposed algorithms.

To support reproducibility of our approach, we open source the fuzzing logic, especially the implementation of GRIMOIRE as well as its interaction with REDQUEEN at https://github.com/RUB-SysSec/grimoire.

## 5 Evaluation

We evaluate our prototype implementation GRIMOIRE to answer the following research questions.

**RQ 1** How does GRIMOIRE compare to other state-of-the-art bug finding tools?

**RQ 2** Is our approach useful even when proper grammars are available?

**RQ 3** How does our approach improve the performance on targets that require highly structured inputs?

**RQ 4** How does our approach perform compared to other grammar inference techniques for the purpose of fuzzing?

**RQ 5** How do our mutators impact fuzzing performance?

**RQ 6** Can GRIMOIRE identify new bugs in real-world applications?

To answer these questions, we perform three individual experiments. First, we evaluate the coverage produced by various fuzzers on a set of real-world target programs. In the second experiment, we analyze how our techniques can be combined with grammar-based fuzzers for mutual improvements. Finally, we use GRIMOIRE to uncover a set of vulnerabilities in real-world target applications.

## 5.1 Measurement Setup

All experiments are performed on an Ubuntu Server 16.04.2 LTS with an Intel i7-6700 processor with 4 cores and 24 GiB of RAM. Each tool is evaluated over 12 runs for 48 hours to obtain statistically meaningful results. In addition to other statistics, we also measure the effect size by calculating the difference in the median of the number of basic blocks found in each run. Additionally, we perform a Mann Whitney U test (using `scipy` 1.0 [38]) and report the resulting *p*-values. All experiments are performed with the tool being pinned to a dedicated CPU in single-threaded mode. Tools other than GRIMOIRE and REDQUEEN require source-code access; we use the fast *clang*-based instrumentation in these cases. Additionally, to ensure a fair evaluation, we provide each fuzzer with a dictionary containing the strings found inside of the target binary. In all cases, except NAUTILUS (which crashed on larger bitmaps), we increase the bitmap size from $2^{16}$ to $2^{19}$. This is necessary since we observe more collisions in the global coverage map for large targets which causes the fuzzer to discard new coverage. For example, in `SQLite` (1.9 MiB), 14% of the global coverage map entries collide [66]. Since we deal with even larger binaries such as `PHP` which is nearly 19 MiB, the bitmap fills up quickly. Based on our empirical evaluation, we observed that $2^{19}$ is the smallest sufficient size that works for all of our target binaries.

Furthermore, we disable the so-called *deterministic stage* [66]. This is motivated by the observation that these deterministic mutations are not suited to find new coverage considering the nature of highly structured inputs. Finally—if not stated otherwise—we use the same uninformed seed that the authors of REDQUEEN used for their experiments: `"ABC...XYZabc...xyz012...789!"$...~+*"`.

As noted by Aschermann et al. [3], there are various definitions of a basic block. Fuzzers such as AFL change the number of basic blocks in a program. Thus, to enable a fair comparison in our experiments, we measure the coverage produced by each fuzzer on the same uninstrumented binary. Therefore, the numbers of basic blocks found and reported in our paper might differ from other papers. However, they are consistent within all of our experiments.

For our experiments, we select a diverse set of target programs. We use four scripting language interpreters (`mruby-1.4.1` [41], `php-7.3.0` [57], `lua-5.3.5` [36] and `JavaScriptCore, commit "f1312"` [1]) a compiler (`tcc-0.9.27` [6]), an assembler (`nasm-2.14.02` [56]), a database (`sqlite-3.25` [31]), a parser (`libxml-2.9.8` [59]) and an SMT solver (`boolector-3.0.1` [44]). We select these four scripting language interpreters so that we can directly compare the results to NAUTILUS. Note that our choice of targets is additionally governed by architectural limitations of REDQUEEN which GRIMOIRE is based on. REDQUEEN uses Virtual Machine Introspection (VMI) to transfer the target binary—including all of its dependencies—into the Virtual Machine (VM). The maximum transfer size using VMI in REDQUEEN is set to 64 MiB. Programs such as `Python` [49], `GCC` [18], `Clang` [40], `V8` [24] and `SpiderMonkey` [43] exceed our VMI limitation; thus, we can not evaluate them. We select an alternative set of target binaries that are large enough but at the same time do not exceed our 64 MiB transfer size limit. Hence, we choose `JavaScriptCore` over `V8` and `SpiderMonkey`, `mruby` over `ruby` and `TCC` over `GCC` or `Clang`. Finally, we tried to evaluate GRIMOIRE with `ChakraCore` [42]. However, `ChakraCore` fails to start inside of the REDQUEEN Virtual Machine for unknown reasons. Still, GRIMOIRE performs well on large targets such as `JavaScriptCore` and `PHP`.

## 5.2 State-of-the-Art Bug Finding Tools

To answer **RQ 1**, we perform 12 runs on eight targets using GRIMOIRE and four state-of-the-art bug finding tools. We choose AFL (version 2.52b) because it is a well-known fuzzer and a good baseline for our evaluation. We select QSYM (commit "6f00c3d") and ANGORA (commit "6ff81c6"), two state-of-the-art hybrid fuzzers which employ different program analysis techniques, namely symbolic execution and taint tracking. Finally, we choose REDQUEEN as a state-of-the-art coverage-guided fuzzer, which is also the baseline of GRIMOIRE. As a consequence, we are able to directly observe the improvements of our method. Note that we could not compile `libxml` for ANGORA instrumentation. Therefore, ANGORA is missing in the `libxml` plot.

The results of our coverage measurements are shown in Figure 4. As we can see, in all cases GRIMOIRE provides a significant advantage over the baseline (unmodified REDQUEEN). Surprisingly, in most cases, neither ANGORA, REDQUEEN, nor QSYM seem to have a significant edge over plain AFL. This can be explained by the fact that REDQUEEN and ANGORA mostly aim to overcome certain "magic byte" fuzzing roadblocks. Similarly, QSYM is also effective to solve these roadblocks. Since we provide a dictionary with strings from the target binary to each fuzzer, these roadblocks become much less common. Thus, the techniques introduced in ANGORA, REDQUEEN and QSYM are less relevant given the seeds provided to the fuzzers. However, in the case of `TCC`, we can observe that providing the strings dictionary does not help AFL. Therefore, we believe that ANGORA and REDQUEEN

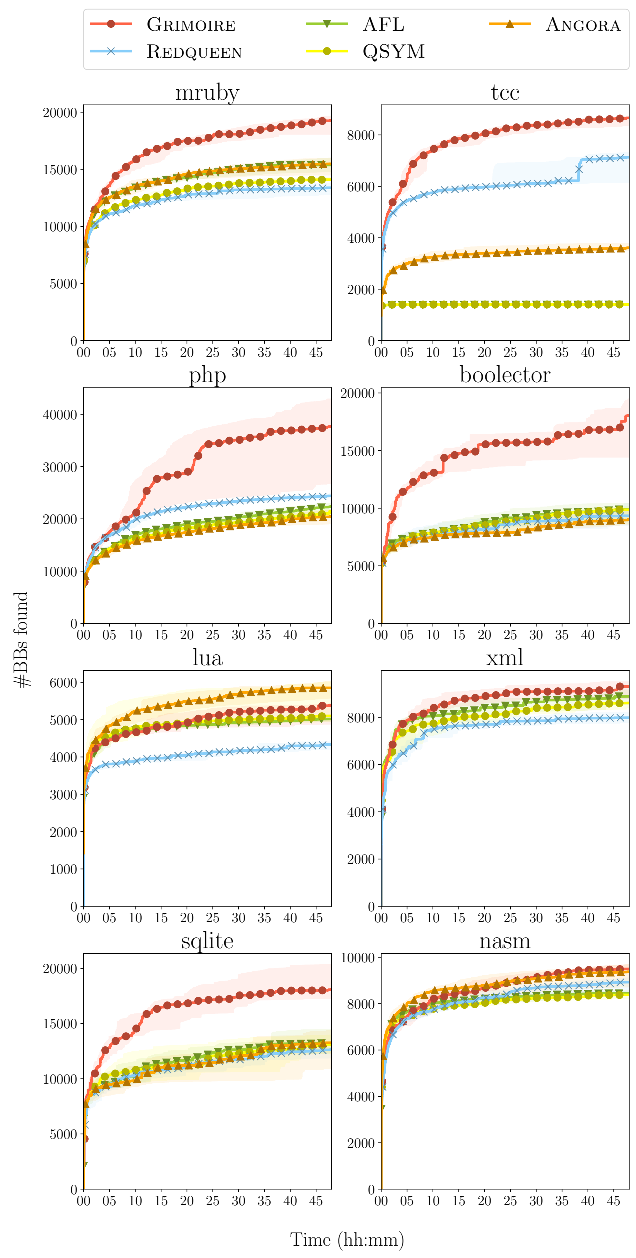


Figure 4: The coverage (in basic blocks) produced by various tools over 12 runs for 48h on various targets. Displayed are the median and the 66.7% intervals.

find strings that are not part of the dictionary and thus outperform AFL.

A complete statistical description of the results is given in the appendix (Table 7). We perform a confirmatory statistical analysis on the results, as shown in Table 2. The results show that in all but two cases (`Lua` and `NASM`), GRIMOIRE offers relevant and significant improvements over all state-of-the-art alternatives. On average, it finds nearly 20% more coverage than the second best alternative.

Table 2: Confirmatory data analysis of our experiments. We compare the coverage produced by GRIMOIRE against the best alternative. The effect size is the difference of the medians in basic blocks. In most experiments, the effect size is relevant and the changes are highly significant: it is typically multiple orders of magnitude smaller than the usual bound of $p < 5.0E-02$ (bold).

| Target | Best Alternative | Effect Size ($\Delta = \bar{A} - \bar{B}$) | Effect Size in % of Best | p-value |
|---|---|---|---|---|
| `mruby` | ANGORA | 3685 | 19.3% | **1.8E-05** |
| `TCC` | REDQUEEN | 1952 | 22.6% | **7.8E-05** |
| `PHP` | REDQUEEN | 11238 | 31.6% | **1.8E-05** |
| `Boolector` | AFL | 7671 | 43.9% | **1.8E-05** |
| `Lua` | ANGORA | -478 | -8.2% | **4.5E-04** |
| `libxml` | AFL | 308 | 3.4% | **1.8E-02** |
| `SQLite` | ANGORA | 4846 | 26.8% | **1.8E-05** |
| `NASM` | ANGORA | 272 | 2.9% | 9.7E-02 |

`Lua` accepts both source files (text) as well as byte code. GRIMOIRE can only make effective mutations in the domain of language features and not the bytecode. However, other fuzzers can perform on both; this is why ANGORA outperforms GRIMOIRE on this target. It is worth mentioning that GRIMOIRE outperforms REDQUEEN, the baseline on top of which our approach is implemented.

To partially answer **RQ 1**, we showed that in terms of *code coverage*, GRIMOIRE outperforms other state-of-the-art bug finding tools (in most cases). Second, to answer **RQ 3**, we demonstrated that GRIMOIRE significantly improves the performance on targets with highly structured inputs when compared to our baseline (REDQUEEN).

## 5.3 Grammar-based Fuzzers

Generally, we expect grammar-based fuzzers to have an edge over grammar inference fuzzers like GRIMOIRE since they have access to a manually crafted grammar. To quantify this advantage, we evaluate GRIMOIRE against current grammar-based fuzzers. To this end, we choose NAUTILUS (commit "dd3554a"), a state-of-the-art coverage-guided fuzzer, since it can fuzz a wide variety of targets if provided with a hand-written grammar. We evaluate on the targets used in NAUTILUS' experiments, `mruby`, `PHP` and `Lua`, as their grammars are available. Unfortunately, GRIMOIRE is not capable of running `ChakraCore`, the fourth target NAUTILUS was evaluated on; thus, we replace it by `JavaScriptCore` and use NAUTILUS' JavaScript grammar. We observed that the original version of NAUTILUS had some timeout problems during fuzzing where the timeout detection did not work properly. We fixed this for our evaluation.

For each of the four targets, we perform an experiment with the same setup as the first experiment (again, 12 runs for 48 hours). The results are shown in Figure 5. As expected, our completely automated method is defeated in most cases by NAUTILUS since it uses manually fine-tuned grammars.

Surprisingly, in the case of mruby, we find that GRIMOIRE is able to outperform even NAUTILUS.

To evaluate whether GRIMOIRE is still useful in scenarios where a grammar is available, we perform another experiment. We extract the corpus produced by NAUTILUS after half of the time (i. e., 24 hours) and continue to use GRIMOIRE for another 24 hours using this seed corpus. For these *incremental runs*, we reduce GRIMOIRE's upper bound for input generalization to 2,048 bytes; otherwise, our fuzzer would mainly spend time in the generalization phase since NAUTILUS produces very large inputs. The results are displayed in Figure 5 (incremental). This experiment demonstrates that even despite manual fine-tuning, the grammar often contains blind spots, where an automated approach such as ours can infer the implicit structure which the program expects. This structure may be quite different from the specified grammar. As Figure 5 shows, by using the corpus created by NAUTILUS, GRIMOIRE surpasses NAUTILUS individually in all cases (**RQ 2**). A confirmatory statistical analysis of the results is presented in Table 3. In three cases, GRIMOIRE is able to improve upon hand written grammars by nearly 10%.

Table 3: Confirmatory data analysis of our experiment. We compare the coverage produced by GRIMOIRE against NAUTILUS with hand written grammars. The effect size is the difference of the medians in basic blocks in the incremental experiment. In three experiments, the effect size is relevant and the changes are highly significant (marked bold, p < 5.0E-02). Note that we abbreviate JavaScriptCore with JSC.

| Target | Best Alternative | Effect Size ($\Delta = \bar{A} - \bar{B}$) | Effect Size in % of Best | p-value |
|---|---|---|---|---|
| mruby | NAUTILUS | 2025 | 10.0% | **1.8E-05** |
| Lua | NAUTILUS | 553 | 5.2% | 5.0E-02 |
| PHP | NAUTILUS | 5465 | 9.3% | **3.6E-03** |
| JSC | NAUTILUS | 15445 | 11.0% | **1.8E-05** |

Additionally, we intended to compare GRIMOIRE against CODEALCHEMIST and JSFUNFUZZ, two other state-of-the art grammar-based fuzzers which specialize on JavaScript engines. Although these two fuzzers are not coverage-guided—making a fair evaluation challenging—we consider the comparison of specialized JavaScript grammar-based fuzzers to general-purpose grammar-based fuzzers as interesting. Unfortunately, JSFUNFUZZ was not working with JavaScriptCore out of the box as it is specifically tailored to SpiderMonkey. Since it requires significant modifications to run on JavaScriptCore, we considered the required engineering effort to be out of scope for this paper. On the other hand, CODEALCHEMIST requires an extensive seed corpus of up to 60,000 valid JavaScript files—which were not released together with the source files. We tried to replicate the seed corpus as described by the authors of CODEALCHEMIST. However, despite the authors' kind help, we were unable to run CODEALCHEMIST with our corpus.

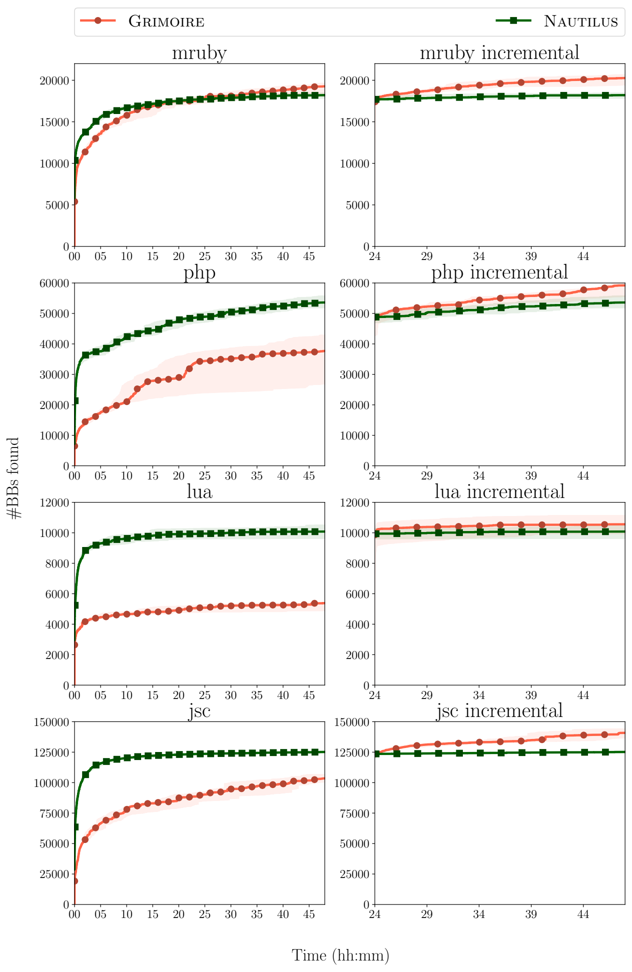


Figure 5: The coverage (in basic blocks) produced by GRIMOIRE and NAUTILUS (using the hand written grammars of the authors of NAUTILUS) over 12 runs at 48 h on various targets. The incremental plots show how running NAUTILUS for 48h compares to running NAUTILUS for the first 24h and then continue fuzzing for 24h with GRIMOIRE. Displayed are the median and the 66.7% confidence interval.

Overall, these experiments confirm our assumption that grammar-based fuzzers such as NAUTILUS have an edge over grammar inference fuzzers like GRIMOIRE. However, deploying our approach on top of a grammar-based fuzzer (incremental runs) increases code coverage. Therefore, we partially respond to **RQ 1** and provide an answer to **RQ 2** by stating that GRIMOIRE is a valuable addition to current fuzzing techniques.

## 5.4 Grammar Inference Techniques

To answer **RQ 4**, we compare our approach to other grammar inference techniques in the context of fuzzing. Existing work in this field includes GLADE, AUTOGRAM and PYGMALION. However, since PYGMALION targets only Python and AUTOGRAM only Java programs, we cannot evaluate

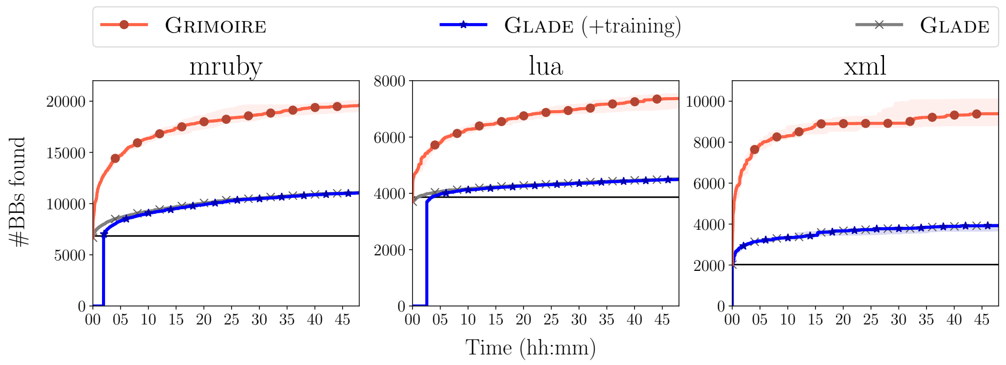


Figure 6: Comparing GRIMOIRE against GLADE (median and 66.7% interval). In the plot for GLADE +Training, we include the training time that glade used. For comparison, we also include plots where we omit the training time. The horizontal bar displays the coverage produced by the seed corpus that GLADE used during training.

them as GRIMOIRE only supports targets that can be traced with Intel-PT (since REDQUEEN heavily depends on it).

Therefore, for this evaluation, we use GLADE (commit "b9ef32e"), a state-of-the-art grammar inference tool. It operates in two stages. Given a program as black-box oracle as well as a corpus of valid input samples, it learns a grammar in the first stage. In the second stage, GLADE uses this grammar to produce inputs that can be used for fuzzing. GLADE does not generate a continuous stream of inputs, hence we modified it to provide such capability. We then use these inputs to measure the coverage achieved by GLADE in comparison to GRIMOIRE. Note that due to the excessive amount of inputs produced by GLADE, we use a corpus minimization tool—`afl-cmin`—to identify and remove redundant inputs before measuring the coverage [66].

Note, we have to extend GLADE for each target that is not natively supported and must manually create a valid seed corpus. For this reason, we restrict ourselves to the three targets `libxml`, `mruby` and `Lua`. From these, `libxml` is the only one that was also used in GLADE's evaluation. Therefore, we are able to re-use their provided corpus for this target. We choose the other two since we want to achieve comparability with regards to previous experiments.

To allow for a fair comparison, we provide the same corpus to GRIMOIRE. Again, we repeat all experiments 12 times for 48 hours each. The results of this comparison are depicted in Figure 6. Note that this figure includes two different experiments of GLADE. In the first experiment, we include the time GLADE spent on training into the measurement while for the second measurement, GLADE is provided the advantage of concluding the training stage before measurement is started for the fuzzing process. As can be seen in Figure 6, GRIMOIRE significantly outperforms GLADE on all targets for both experiments. Similar to earlier experiments, we perform a confirmatory statistical analysis. The results are displayed in Table 4; they are in all cases relevant and statistically significant. If we consider only the new coverage found (beyond what is already contained in the training set), we are able to outperform GLADE by factors from two to five. We therefore conclude in response to **RQ 4** that we significantly exceed comparative grammar inference approaches in the context of fuzzing.

We designed another experiment to evaluate whether GLADE's automatically inferred grammar can be used for NAUTILUS and how it performs compared to hand written grammars. However, GLADE does not use the grammar directly but remembers how the grammar was produced from the provided test cases and uses the grammar only to apply local mutations to the input. Unfortunately, as a consequence, their grammar contains multiple unproductive rules, thus preventing their usage in NAUTILUS.

Table 4: Confirmatory data analysis of our experiments. We compare the coverage produced by GRIMOIRE against GLADE. The effect size is the difference of the medians in basic blocks. In all experiments, the effect size is relevant and the changes are highly significant: it is multiple orders of magnitude smaller than the usual bound of $p < 5.0E{-}02$ (bold).

| Target | Best Alternative | Effect Size ($\Delta = \bar{A} - \bar{B}$) | Effect Size in % of Best | p-value |
|---|---|---|---|---|
| `mruby` | GLADE | 8546 | 43.6% | **9.1E-05** |
| `Lua` | GLADE | 2775 | 38.1% | **9.1E-05** |
| `libxml` | GLADE | 5213 | 57.2% | **9.1E-05** |

## 5.5 Mutations Statistic

During the aforementioned experiments, we also collected various statistics on how effective different mutators are. We measured how much time was spent using GRIMOIRE's different mutation strategies as well as how many of the inputs were found by each strategy. This allows us to rank mutation strategies based on the number of new paths found per time used. The strategies include a havoc stage, REDQUEEN's Input-to-State-based mutation stage and our structural mutation stage. The times for our structural mutators include the generalization process (including the necessary minimization that also benefits the other mutators).

As Table 5 shows, our structural mutators are competitive with other mutators, which answers **RQ 5**. As the coverage results in Figure 4 show, the mutators are also able to uncover paths that would not have been found otherwise.

Table 5: Statistics for each of GRIMOIRE's mutation strategies (i. e., our structured mutations, REDQUEEN's Input-to-State-based mutations and havoc). For every target evaluated we list the total number of inputs found by a mutation, the time spent on this strategy and the ratio of inputs found per minute.

| Mutation | Target | #Inputs | Time Spent (min) | #Inputs/Min |
|---|---|---|---|---|
| Structured | `mruby` | 9040 | 1531.18 | 5.90 |
| | `PHP` | 27063 | 2467.17 | **10.97** |
| | `Lua` | 2849 | 2064.49 | 1.38 |
| | `SQLite` | 5933 | 1325.26 | **4.48** |
| | `TCC` | 6618 | 2271.03 | 2.91 |
| | `Boolector` | 3438 | 2399.85 | 1.43 |
| | `libxml` | 4883 | 2001.38 | 2.44 |
| | `NASM` | 12696 | 1955.42 | **6.49** |
| | `JavaScriptCore` | 38465 | 2460.95 | **15.63** |
| Input-to-State | `mruby` | 814 | 268.23 | 3.03 |
| | `PHP` | 902 | 111.46 | 8.09 |
| | `Lua` | 530 | 307.12 | 1.73 |
| | `SQLite` | 603 | 768.72 | 0.78 |
| | `TCC` | 1020 | 118.23 | **8.63** |
| | `Boolector` | 325 | 102.87 | **3.16** |
| | `libxml` | 967 | 359.03 | 2.69 |
| | `NASM` | 1329 | 213.84 | 6.22 |
| | `JavaScriptCore` | 400 | 82.76 | 4.83 |
| Havoc | `mruby` | 2010 | 339.03 | **5.93** |
| | `PHP` | 2546 | 278.21 | 9.15 |
| | `Lua` | 1684 | 492.99 | **3.42** |
| | `SQLite` | 1827 | 742.13 | 2.46 |
| | `TCC` | 2514 | 484.73 | 5.19 |
| | `Boolector` | 956 | 373.85 | 2.56 |
| | `libxml` | 2173 | 504.86 | **4.30** |
| | `NASM` | 2876 | 678.59 | 4.24 |
| | `JavaScriptCore` | 3800 | 279.62 | 13.59 |

## 5.6 Real-World Bugs

We use GRIMOIRE on a set of different targets to observe whether it is able to uncover previously unknown bugs (**RQ 6**). To this end, we manually triaged bugs found during our evaluation. As illustrated in Table 6, GRIMOIRE found more bugs than all other tools in the evaluation combined. We responsibly disclosed all of them to the vendors. For these, 11 CVEs were assigned. Note that we found a large number of bugs that did not lead to assigned CVEs. This is partially because projects such as `PHP` do not consider invalid inputs as security relevant, even when custom scripts can trigger memory corruption. We conclude **RQ 6** by finding that GRIMOIRE is indeed able to uncover novel bugs in real-world applications.

## 6 Discussion

The methods introduced in this paper produce significant performance gains on targets that expect highly structured inputs without requiring any expert knowledge or manual work. As we have shown, GRIMOIRE can also be used to support grammar-based fuzzers with well-tuned grammars but

Table 6: Overview of submitted bugs and CVEs. Fuzzers which did not find the bug during our evaluation are denoted by ✗, while those who did are marked by ✓. We indicate targets not evaluated by a specific fuzzer with '-'. We abbreviate Use-After-Free (UAF), Out-of-Bounds (OOB) and Buffer Overflow (BO).

| Target | CVE | Type | GRIMOIRE | REDQUEEN | AFL | QSYM | ANGORA | NAUTILUS |
|---|---|---|---|---|---|---|---|---|
| `PHP` | | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| `PHP` | | OOB-read | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ |
| `PHP` | | OOB-read | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| `PHP` | | OOB-read | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| `TCC` | 2018-20374 | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| `TCC` | 2018-20375 | OOB-write | ✓ | ✓ | ✗ | ✗ | ✗ | - |
| `TCC` | 2018-20376 | OOB-write | ✓ | ✓ | ✗ | ✗ | ✗ | - |
| `TCC` | 2019-12495 | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| `TCC` | 2019-9754 | OOB-write | ✓ | ✓ | ✗ | ✗ | ✗ | - |
| `TCC` | | OOB-write | ✗ | ✓ | ✗ | ✗ | ✗ | - |
| `Boolector` | 2019-7559 | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| `Boolector` | 2019-7560 | UAF-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| `NASM` | 2019-8343 | UAF-write | ✓ | ✓ | ✗ | ✗ | ✗ | - |
| `NASM` | | OOB-write | ✓ | ✗ | ✓ | ✗ | ✗ | - |
| `NASM` | | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| `NASM` | | OOB-write | ✓ | ✗ | ✗ | ✗ | ✗ | - |
| `NASM` | | OOB-write | ✓ | ✗ | ✓ | ✗ | ✗ | - |
| `NASM` | | OOB-write | ✗ | ✗ | ✓ | ✗ | ✗ | - |
| `gnuplot` | 2018-19490 | BO | ✓ | - | - | - | - | - |
| `gnuplot` | 2018-19491 | BO | ✓ | - | - | - | - | - |
| `gnuplot` | 2018-19492 | BO | ✓ | - | - | - | - | - |

cannot outperform them on their own. In contrast to similar methods, our approach does not rely on complex primitives such as symbolic execution or taint tracking. Therefore, it can easily be integrated into existing fuzzers. Additionally, since GRIMOIRE is based on REDQUEEN, it can be used on a wide variety of binary-only targets, ranging from userland programs to operating system kernels.

Despite all advantages, our approach has significant difficulties with more syntactically complex constructs, such as matching the ID of opening and closing tags in XML or identifying variable constructs in scripting languages. For instance, while GRIMOIRE is able to produce nested inputs such as "`<a><a><a>FOO</a></a></a>`", it struggles to generalize "`<a>□</a>`" to the more unified representation "<$\boxed{A}$>□</$\boxed{B}$>" with the constraint $A = B$. A solution for such complex constructs could be the following generalization heuristic: (i) First, we record the new coverage for the current input. (ii) We then change only a single occurrence of a substring in our input and record its new coverage. For instance, consider that we replace a single occurrence of "a" by "b" in "`<a><a><a>FOO</a></a></a>`" and obtain "`<b><a><a>FOO</a></a></a>`". This change results in an invalid XML tag which leads to different coverage compared to the one observed in (i). (iii) Finally, we change multiple instances of the same substring and compare the new coverage of the modified input with the one obtained in (i). If we achieved the same new coverage in (iii) and (i), we can assume that the modified instances of the same substring are related to each other. For example, we replace multiple occurrences of “a” with “b” and obtain “<b><a><a>FOO</a></a></b>”. In this example, the coverage is the same as for the original input since the XML remains syntactically correct.

Similarly, our generalization approach might be too coarse in many places. Obtaining more precise rules would help uncovering deeper parts of the target application in cases where multiple valid statements have to be produced. Consider, for instance, a scripting language interpreter such as the ones used in our evaluation. Certain operations might require a number of constructors to be successfully called. For example, it might be necessary to get a valid path object to obtain a file object that can finally be used to perform a read operation. A more precise representation would be highly useful in such cases. One could try to infer whether a combination is “valid” by checking if the combination of two inputs exercises the combination of the new coverage introduced by both inputs. For instance, assume that input “a□b” triggers the coverage bytes 7 and 10 and that input “□=□” triggers coverage byte 20. Then, a combination of these two inputs such as “□a□=□b” could trigger the coverage bytes 7, 10 and 20. Using this information, it might be possible to infer more precise grammar descriptions and thus generate inputs that are closer to the target’s semantics than it is currently possible in GRIMOIRE. While this approach would most likely further reduce the gap between hand-written grammars and inferred grammars, well-designed hand-written grammars will always have an edge over fuzzers with no prior knowledge: any kind of inference algorithm first needs to uncover structures before the obtained knowledge can be used. A grammar-based fuzzer has no such disadvantage. If available, human input can improve the results of grammar inference or steer its direction. An analyst can provide a partial grammar to make the grammar-fuzzer focus on a specific interesting area and avoid exploring paths that are unlikely to contain bugs. Therefore, GRIMOIRE is useful if the grammar is unknown or under-specified but cannot be considered a full replacement for grammar-based fuzzers.

## 7 Related Work

A significant number of approaches to improve the performance of different fuzzing strategies has been proposed over time. Early on, fuzzers typically did not observe the inner workings of the target application, yet different approaches were proposed to improve various aspects of fuzzers: different mutation strategies were evaluated [14,29], the process of selecting and scheduling of seed inputs was analyzed [11,51,61] and, in some cases, even learned language models were used to improve the effectiveness of fuzzing [22,27]. After the publication of AFL [65], the research focus shifted towards coverage-guided fuzzing techniques. Similarly to the previous work on blind fuzzing, each individual component of AFL was put under scrutiny. For example, AFLFAST [8] and AFLGo [7] proposed scheduling mechanisms that are better suited to some circumstances. Both, COLLAFL [16] and InsTrim [35], enhanced the way in which coverage is generated and stored to reduce the amount of memory needed. Other publications improved the ways in which coverage feedback is collected [23,53,55,62]. To advance the ability of fuzzers to overcome constraints that are hard to guess, a wide array of techniques were proposed. Commonly, different forms of symbolic execution are used to solve these challenging instances [9, 10]. In most of these cases, a restricted version of symbolic execution (concolic execution) is used [19–21, 26, 54, 60]. To further improve upon these techniques, DigFuzz [67] provides a better scheduling for inputs to the symbolic executor. Sometimes, instead of using these heavy-weight primitives, more lightweight techniques such as taint tracking [12, 17, 26, 50], patches [3, 13, 47, 60] or instrumentation [3, 39] are used to overcome the same hurdles.

While these improvements generally work very well for binary file formats, many modern target programs work with highly structured data. To target these programs, generational fuzzing is typically used. In such scenarios, the user can often provide a grammar. In most cases, fuzzers based on this technique are blind fuzzers [14,33,45,52,63].

Recent projects such as AFLSMART [48], NAUTILUS [2] and ZEST [46] combined the ideas of generational fuzzing with coverage guidance. CODEALCHEMIST [28] even ventures beyond syntactical correctness. To find novel bugs in mature JavaScript interpreters, it tries to automatically craft syntactically and semantically valid inputs by recombining input fragments based on inferred types of variables. All of these approaches require a good format specification and—in some cases—good seed corpora. CODEALCHEMIST even needs access to a specialized interpreter for the target language to trace and infer type annotations. In contrast, our approach has no such preconditions and is thus easily integrable into most fuzzers.

Finally, to alleviate some of the disadvantages that the mentioned grammar-based strategies have, multiple approaches were developed to automatically infer grammars for given programs. GLADE [5] can systematically learn an approximation to the context-free grammars parsed by a program. To learn the grammar, it needs an oracle that can answer whether a given input is valid or not as well as a small set of valid inputs. Similar techniques are used by PYGMALION [25] and AUTOGRAM [34]. However, both techniques directly learn from the target application without requiring a modified version of the target. AUTOGRAM still needs a large set of inputs to trace, while PYGMALION can infer grammars based solely on the target application. Additionally, both approaches require complex analysis passes and even symbolic execution to produce grammars. These techniques cannot easily be scaled to large binary applications. Finally, all three approaches are computationally expensive.

## 8 Conclusion

We developed and demonstrated the first fully automatic algorithm that integrates large-scale structural mutations into the fuzzing process. In contrast to other approaches, we need no additional modifications or assumptions about the target application. We demonstrated the capabilities of our approach by evaluating our implementation called GRIMOIRE against various state-of-the-art coverage-guided fuzzers. Our evaluation shows that we outperform other coverage-guided fuzzers both in terms of coverage and the number of bugs found. From this observation, we conclude that it is possible to significantly improve the fuzzing process in the absence of program input specifications. Furthermore, we conclude that even when a program input specification is available, our approach is still useful when it is combined with a generational fuzzer.

## Acknowledgements

We would like to thank our shepherd Deian Stefan and the anonymous reviewers for their valuable comments and suggestions. Furthermore, we would like to thank Moritz Contag, Thorsten Eisenhofer, Joel Frank, Philipp Görz and Maximilian Golla for their valuable feedback. This work was supported by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States - EXC 2092 CASA. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 786669 (ReAct). This paper reflects only the authors' view. The Research Executive Agency is not responsible for any use that may be made of the information it contains.

## References

[1] APPLE INC. JavaScriptCore. https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore.

[2] ASCHERMANN, C., FRASSETTO, T., HOLZ, T., JAUERNIG, P., SADEGHI, A.-R., AND TEUCHERT, D. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)* (2019).

[3] ASCHERMANN, C., SCHUMILO, S., BLAZYTKO, T., GAWLIK, R., AND HOLZ, T. REDQUEEN: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)* (2019).

[4] BASTANI, O., SHARMA, R., AIKEN, A., AND LIANG, P. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2017).

[5] BASTANI, O., SHARMA, R., AIKEN, A., AND LIANG, P. Synthesizing program input grammars. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2017).

[6] BELLARD, F. TCC: Tiny C compiler. https://bellard.org/tcc/.

[7] BÖHME, M., PHAM, V.-T., NGUYEN, M.-D., AND ROYCHOUDHURY, A. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[8] BÖHME, M., PHAM, V.-T., AND ROYCHOUDHURY, A. Coverage-based greybox fuzzing as Markov chain. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[9] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[10] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy* (2012).

[11] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy* (2015).

[12] CHEN, P., AND CHEN, H. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy* (2018).

[13] DREWRY, W., AND ORMANDY, T. Flayer: Exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies* (2007), USENIX Association.

[14] EDDINGTON, M. Peach fuzzer: Discover unknown vulnerabilities. https://www.peach.tech/.

[15] FREE SOFTWARE FOUNDATION. GNU Bison. https://www.gnu.org/software/bison/.

[16] GAN, S., ZHANG, C., QIN, X., TU, X., LI, K., PEI, Z., AND CHEN, Z. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy* (2018).

[17] GANESH, V., LEEK, T., AND RINARD, M. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)* (2009).

[18] GNU PROJECT. GCC, the GNU compiler collection. https://gcc.gnu.org/.

[19] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2008).

[20] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2005).

[21] GODEFROID, P., LEVIN, M. Y., MOLNAR, D. A., ET AL. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)* (2008).

[22] GODEFROID, P., PELEG, H., AND SINGH, R. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), pp. 50–59.

[23] GOODMAN, P. Shin GRR: Make fuzzing fast again. https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/.

[24] GOOGLE LLC. V8. https://v8.dev/.

[25] GOPINATH, R., MATHIS, B., HÖSCHELE, M., KAMPMANN, A., AND ZELLER, A. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289* (2018).

[26] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium* (2013).

[27] HAN, H., AND CHA, S. K. IMF: Inferred model-based fuzzer. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[28] HAN, H., OH, D., AND CHA, S. K. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Symposium on Network and Distributed System Security (NDSS)* (2019).

[29] HELIN, A. A general-purpose fuzzer. `https://github.com/aoh/radamsa`.

[30] HEX-RAYS. IDA pro. `https://www.hex-rays.com/products/ida/`.

[31] HIPP, D. R. SQLite. `https://www.sqlite.org/index.html`.

[32] HOCEVAR, S. zzuf. `https://github.com/samhocevar/zzuf`.

[33] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with code fragments. In *USENIX Security Symposium* (2012).

[34] HÖSCHELE, M., AND ZELLER, A. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (2016).

[35] HSU, C.-C., WU, C.-Y., HSIAO, H.-C., AND HUANG, S.-K. INSTRIM: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research* (2018).

[36] IERUSALIMSCHY, R., CELES, W., AND DE FIGUEIREDO, L. H. Lua. `https://www.lua.org/`.

[37] JOHNSON, S. Yacc: Yet another compiler-compiler. `http://dinosaur.compilertools.net/yacc/`.

[38] JONES, E., OLIPHANT, T., AND PETERSON, P. Scipy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–.

[39] LI, Y., CHEN, B., CHANDRAMOHAN, M., LIN, S.-W., LIU, Y., AND TIU, A. Steelix: Program-state based binary fuzzing. In *Joint Meeting on Foundations of Software Engineering* (2017).

[40] LLVM PROJECT. Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`.

[41] MATSUMOTO, Y. mruby. `http://mruby.org/`.

[42] MICROSOFT. ChakraCore. `https://github.com/Microsoft/ChakraCore`.

[43] MOZILLA FOUNDATION / MOZILLA CORPORATION. SpiderMonkey. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

[44] NIEMETZ, A., PREINER, M., AND BIERE, A. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation 9* (2015), 53–58.

[45] OPENRCE. Sulley: A pure-python fully automated and unattended fuzzing framework. `https://github.com/OpenRCE/sulley`.

[46] PADHYE, R., LEMIEUX, C., SEN, K., PAPADAKIS, M., AND TRAON, Y. L. Zest: Validity fuzzing and parametric generators for effective random testing. *arXiv preprint arXiv:1812.00078* (2018).

[47] PENG, H., SHOSHITAISHVILI, Y., AND PAYER, M. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy* (2018).

[48] PHAM, V.-T., BÖHME, M., SANTOSA, A. E., CĂCIULESCU, A. R., AND ROYCHOUDHURY, A. Smart greybox fuzzing, 2018.

[49] PYTHON SOFTWARE FOUNDATION. Python. `https://www.python.org/`.

[50] RAWAT, S., JAIN, V., KUMAR, A., COJOCAR, L., GIUFFRIDA, C., AND BOS, H. VUzzer: Application-aware evolutionary fuzzing. In *Symposium on Network and Distributed System Security (NDSS)* (Feb. 2017).

[51] REBERT, A., CHA, S. K., AVGERINOS, T., FOOTE, J. M., WARREN, D., GRIECO, G., AND BRUMLEY, D. Optimizing seed selection for fuzzing. In *USENIX Security Symposium* (2014).

[52] RUDERMAN, J. Introducing jsfunfuzz. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz` (2007).

[53] SCHUMILO, S., ASCHERMANN, C., GAWLIK, R., SCHINZEL, S., AND HOLZ, T. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *USENIX Security Symposium* (2017).

[54] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)* (2016).

[55] SWIECKI, R. Security oriented fuzzer with powerful analysis options. `https://github.com/google/honggfuzz`.

[56] THE NASM DEVELOPMENT TEAM. NASM. `https://www.nasm.us/`.

[57] THE PHP GROUP. PHP. `http://php.net/`.

[58] VEGGALAM, S., RAWAT, S., HALLER, I., AND BOS, H. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security (ESORICS)* (2016), pp. 581–601.

[59] VEILLARD, DANIEL. The XML C parser and toolkit of Gnome. `http://xmlsoft.org/`.

[60] WANG, T., WEI, T., GU, G., AND ZOU, W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy* (2010).

[61] WOO, M., CHA, S. K., GOTTLIEB, S., AND BRUMLEY, D. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[62] XU, W., KASHYAP, S., MIN, C., AND KIM, T. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[63] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices* (6 2011), vol. 46, ACM, pp. 283–294.

[64] YUN, I., LEE, S., XU, M., JANG, Y., AND KIM, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium* (2018), pp. 745–761.

[65] ZALEWSKI, M. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`.

[66] ZALEWSKI, M. Technical "whitepaper" for afl-fuzz. `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[67] ZHAO, L., DUAN, Y., YIN, H., AND XUAN, J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Symposium on Network and Distributed System Security (NDSS)* (2019).

## A Statistics on Basic Block Coverage

Table 7: Statistics on basic block coverage for tested fuzzers. In the column "Best Coverage", we provide the highest number of basic blocks a run found and the percentage relative to the number of basic blocks obtained from `IDA Pro [30]`.

| Target | Best Coverage (#BBS / %) | Fuzzer | Mean (%) | Median (%) | **Median (#BBs)** | Std Deviation | Skewness | Kurtosis |
|---|---|---|---|---|---|---|---|---|
| `mruby` | 20258 / 70.5% | Grimoire | 66.1% | 66.6% | 19137 | 4.55 | −0.54 | −0.76 |
| | | AFL | 53.7% | 53.4% | 15355 | 4.28 | 0.14 | −0.27 |
| | | Angora | 53.3% | 53.8% | 15452 | 4.87 | 0.17 | −0.96 |
| | | QSym | 49.2% | 49.0% | 14084 | 2.20 | 0.33 | 0.95 |
| | | Redqueen | 45.9% | 46.4% | 13339 | 4.64 | −0.98 | 0.05 |
| `TCC` | 9211 / 77.6% | Grimoire | 71.8% | 72.9% | 8647 | 5.71 | −1.89 | 3.68 |
| | | AFL | 11.8% | 11.8% | 1397 | 3.80 | 1.27 | 1.14 |
| | | Angora | 31.0% | 30.3% | 3600 | 6.51 | 1.01 | 0.06 |
| | | QSym | 11.9% | 11.8% | 1403 | 3.26 | 1.52 | 2.59 |
| | | Redqueen | 56.7% | 56.4% | 6695 | 8.13 | 0.03 | −1.93 |
| `PHP` | 46805 / 27.9% | Grimoire | 20.8% | 21.2% | 35606 | 20.26 | 0.12 | −1.38 |
| | | AFL | 13.2% | 13.3% | 22323 | 3.64 | −0.09 | −0.96 |
| | | Angora | 12.1% | 12.2% | 20501 | 6.39 | −0.37 | −0.58 |
| | | QSym | 12.7% | 12.7% | 21276 | 2.60 | 0.22 | −1.11 |
| | | Redqueen | 14.5% | 14.5% | 24367 | 1.87 | 0.37 | −0.83 |
| `Boolector` | 23207 / 33.1% | Grimoire | 25.2% | 24.9% | 17461 | 16.77 | 0.51 | −0.65 |
| | | AFL | 14.0% | 14.0% | 9790 | 7.46 | 0.30 | −0.57 |
| | | Angora | 13.2% | 12.8% | 8986 | 9.20 | 0.79 | −0.17 |
| | | QSym | 13.7% | 14.0% | 9782 | 6.94 | −0.39 | −1.24 |
| | | Redqueen | 13.3% | 13.3% | 9305 | 9.63 | 0.21 | −1.23 |
| `Lua` | 6205 / 64.1% | Grimoire | 54.4% | 55.2% | 5339 | 6.47 | 0.20 | −0.73 |
| | | AFL | 51.9% | 51.9% | 5016 | 1.61 | 0.84 | −0.15 |
| | | Angora | 59.9% | 60.1% | 5817 | 2.96 | 0.05 | −1.39 |
| | | QSym | 54.8% | 52.6% | 5091 | 9.52 | 1.07 | −0.65 |
| | | Redqueen | 44.5% | 44.4% | 4299 | 2.30 | −0.30 | −1.19 |
| `libxml` | 10437 / 13.2% | Grimoire | 11.7% | 11.6% | 9190 | 5.52 | 0.98 | 0.02 |
| | | AFL | 11.1% | 11.2% | 8881 | 3.40 | −0.39 | −0.92 |
| | | Angora | 0.0% | 0.0% | 0 | nan | 0.00 | −3.00 |
| | | QSym | 10.8% | 10.8% | 8598 | 2.36 | 0.95 | 1.45 |
| | | Redqueen | 10.1% | 10.1% | 7979 | 3.72 | 0.72 | −0.25 |
| `SQLite` | 22031 / 57.1% | Grimoire | 48.6% | 46.8% | 18064 | 9.25 | 0.80 | −0.72 |
| | | AFL | 34.6% | 33.9% | 13072 | 10.02 | 0.60 | −0.34 |
| | | Angora | 33.1% | 34.2% | 13218 | 12.12 | −0.30 | −1.05 |
| | | QSym | 33.4% | 33.6% | 12988 | 10.91 | −0.33 | −0.18 |
| | | Redqueen | 32.3% | 32.6% | 12599 | 4.77 | 0.18 | −0.21 |
| `NASM` | 10015 / 51.1% | Grimoire | 47.7% | 48.4% | 9483 | 7.58 | −2.58 | 5.67 |
| | | AFL | 43.2% | 43.0% | 8442 | 1.68 | 1.07 | 1.09 |
| | | Angora | 46.9% | 47.0% | 9211 | 5.27 | 0.06 | −1.19 |
| | | QSym | 42.1% | 42.6% | 8357 | 4.72 | −1.49 | 2.40 |
| | | Redqueen | 44.9% | 45.5% | 8928 | 4.21 | −0.20 | −0.89 |