Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge

Nils Bars*, Moritz Schloegel*, Tobias Scharnowski*
Nico Schiller*, Thorsten Holz‡

*Ruhr-Universität Bochum
‡CISPA Helmholtz Center for Information Security

Abstract

Today's digital communication relies on complex protocols and specifications for exchanging structured messages and data. Communication naturally involves two endpoints: One generating data and one consuming it. Traditional fuzz testing approaches replace one endpoint, the *generator*, with a fuzzer and rapidly test many mutated inputs on the target program under test. While this fully automated approach works well for loosely structured formats, this does not hold for highly structured formats, especially those that go through complex transformations such as compression or encryption.

In this work, we propose a novel perspective on generating inputs in highly complex formats without relying on heavy-weight program analysis techniques, coarse-grained grammar approximation, or a human domain expert. Instead of mutating the inputs for a target program, we inject faults into the data generation program so that this data is *almost* of the expected format. Such data bypasses the initial parsing stages in the consumer program and exercises deeper program states, where it triggers more interesting program behavior. To realize this concept, we propose a set of compile-time and run-time analyses to mutate the generator in a targeted manner, so that it remains intact and produces semi-valid outputs that satisfy the constraints of the complex format. We have implemented this approach in a prototype called FUZZTRUCTION and show that it outperforms the state-of-the-art fuzzers AFL++, SYMCC, and WEIZZ. FUZZTRUCTION finds significantly more coverage than existing methods, especially on targets that use cryptographic primitives. During our evaluation, FUZZTRUCTION uncovered 151 unique crashes (after automated deduplication). So far, we manually triaged and reported 27 bugs and 4 CVEs were assigned.

1 Introduction

Our modern digital infrastructure is based on well-defined message and data formats, including standards and specifications for data exchange. These systems have at least two endpoints, both of which encode the *domain knowledge* required to handle the communication: One application *generates* the data (hereafter referred to as *generator*), while the other *consumes* it (called *consumer*). For example, various programs generate PDF documents as output and corresponding PDF viewers display the result. Another example is cryptographic libraries, which generate encrypted messages that their corresponding counterparts can process. From a security perspective, the consumer plays a crucial role, as it processes potentially untrusted data and is thus exposed to attacks. Fuzz testing (*fuzzing* for short), a form of randomized testing, has proven helpful in efficiently finding software faults in the input processing of consumer programs. Past advances in fuzzing methods focused on throughput [1–3], effectiveness [4–6], and applicability to new target domains [7–12]. Even after years of research, an unresolved challenge is the effective generation of valid inputs for complex formats, including cryptographic primitives, compression, and other kinds of tricky transformations.

Prior works and industry best practices attempted two approaches to tackle this challenge. First, heavy-weight program analysis techniques, such as symbolic execution and taint analysis, or manual workarounds, have been suggested to solve roadblocks (e. g., checksums or hashes) [5, 6, 13–20]. Unfortunately, these methods do not scale to complex programs. Second, grammar-based fuzzing [21–29] has been investigated to generate inputs of a specific syntactical structure. However, the use of grammars does not address the complexity of applications using complex types of transformations, especially cryptographic primitives and compression.

Instead of using heavy-weight techniques, grammars, or manually bypassing these fuzzing roadblocks, we propose a novel generic approach to generate highly structured and complex inputs for fuzzing in an automated way. More specifically, we propose to take advantage of the domain knowledge that is *already encoded* in the applications that generate data: In traditional fuzzing approaches, the generator is replaced by a fuzzer that passes input directly to the consumer (representing the system under test). On the contrary, we devise a mechanism to *mutate the generator* and then pass its output

to the consumer. The core idea is that this mutated generator produces inputs that *mostly* adhere to the required data format but introduces subtle deviations that may trigger faults in the consumer's processing logic. For example, an application that signs cryptographic certificates knows how to generate a valid signature that can be parsed and verified by any application designed to verify such signatures (e.g., browsers or cryptographic libraries). To enable fuzzing of the signature verification logic, we exploit the fact that the generator implicitly knows how to compute valid signatures. We leverage this knowledge by slightly mutating the generator's code so that the produced output may violate the specification but is technically valid (i.e., a valid signature or encryption).

Randomly flipping instruction bits in the generator would likely not affect its output, and—even worse—it would lead the generator to crash in most cases. To avoid such undesirable cases, we devised a compile-time analysis to identify operations on data and filter out operations that would crash the generator when they are mutated. We also analyze data-flow dependencies to avoid redundant mutations. Then, we identify which parts of the generator actually affect the output for the remaining mutation candidates and focus our mutations on the most promising candidates. Based on these insights, we instrument the generator and just-in-time (JIT)-compile both tracing and mutation mechanisms into it, facilitating efficient fuzzing.

To demonstrate the practical feasibility of the proposed approach, we implement a tool called FUZZTRUCTION. Our method can be used as a stand-alone tool or augment existing fuzzers. In a comprehensive evaluation, we show that our approach not only avoids the shortcomings of traditional approaches for targets that consume complex input formats, but also outperforms fuzzers such as SYMCC [6], WEIZZ [30], and AFL++ [1]: On average, we find 21% more coverage, both for targets that heavily use cryptographic primitives (up to 70% more coverage) and for extensively tested targets (up to 23% more coverage on `objdump`). Beyond coverage, we find more than five times the number of (deduplicated) crashing inputs compared to these fuzzers. During our evaluation, we uncovered 151 unique crashes (after automated deduplication). We manually triaged and reported 27 bugs in a coordinated way to the developers. Up to now, 4 CVEs were assigned.

In summary, we make the following main contributions:

- We propose a novel fuzzing method that automatically leverages the domain knowledge in generator applications to improve fuzzing without relying on advanced and expensive program analysis techniques.
- We demonstrate the generic capabilities of the approach by fuzzing even complex cryptographic procedures, such as the parsing and validation of encrypted RSA keys, automatically and without custom-crafted seeds.
- We implement and evaluate our prototype, called FUZZTRUCTION, against the state-of-the-art fuzzers AFL++,

SYMCC, and WEIZZ. Our results show that our approach achieves significant gains in terms of coverage and number of software faults found.

To foster further research on this topic, we release the source code and evaluation artifacts of FUZZTRUCTION at https://github.com/fuzztruction/fuzztruction.

2   Fuzzing Complex Input Formats

Generally speaking, fuzzing means that we execute a target program with numerous mutated inputs to trigger unexpected behavior, thus revealing faults. To reach deep into the state space of the program under test, fuzzers typically need to generate well-structured inputs. In addition to being well-structured, these inputs also need to account for checksums, compression algorithms, or cryptographic primitives that guard more in-depth processing. In practice, both the efficient identification of logical units within an input format and the subsequent effective resolution of obstacles within these units pose a major challenge. Fuzzers currently attempt to solve these challenges via different approaches, which we outline in the following.

Execution Feedback. Gathering feedback on the execution of the target program for a given input is a well-established technique in modern fuzzers [1, 31, 32]. This feedback provides a measure of input quality and allows a fuzzer to recognize inputs that explore new program behavior. By keeping and further mutating these increasingly well-structured inputs, a fuzzer derives inputs that match the expected formats to a greater degree over time. Gathering execution feedback (and most prominently, coverage feedback) is attractive as it is generically applicable and introduces a low run-time overhead. In fact, after this coverage-guided technique was introduced in AFL [31], fuzzers have been able to explore common, more loosely-structured binary file formats and identify a large number of bugs as a result [33]. Even though it is not directly available to the fuzzer, the execution feedback provides a side channel to the domain knowledge encoded in the program under test.

Grammars. Instead of trying to extract domain knowledge from a target program, a human expert can provide preexisting domain knowledge to the fuzzer. In these approaches, the fuzzer is either provided with information about the input structure of the target program (e.g., a grammar) or a structure-aware logic for input generation is integrated into the fuzzer [21–29]. This precise knowledge about the target format allows the fuzzer to generate inputs that fulfill the target's requirements about the input structure. The main drawback of these approaches lies in the fact that while they allow generating high-quality inputs, they require pre-existing knowledge about the program under test.

Automated Grammar Approximation. Instead of requiring domain knowledge in the form of a grammar a priori, there are techniques to automate the process of generating approximations of the grammar embedded into the target. Different approaches exist which target text-based [34] or binary formats [30, 35]. Both have in common that they try to infer the structure of a provided input by identifying logical units within the data format, e. g., chunks, tokens, or fields. This enables structure-aware mutations that allow the fuzzer to modify, insert, remove, or replace such logical units. Although these techniques sidestep the requirement for crafting the grammar manually and successfully locate logical units, the process of approximating is inherently coarse-grained.

Heavyweight Feedback and Analyses. Beyond the efforts described above, which improve the fuzzer's capability to generate highly structured inputs, an orthogonal line of research has focused on solving typical fuzzing roadblocks, such as checksums or cryptographic primitives, without requiring a grammar. As a more direct way of extracting the domain knowledge encoded in the target program, recent approaches employ sophisticated program analyses and more heavyweight feedback types. These approaches tackle the deficits of (lightweight) feedback-driven fuzzers by aiding the fuzzer in solving constraints within data structures, e. g., via taint tracking [17–19] or concolic/symbolic execution [5, 6, 13–15]. Using taint tracking, the fuzzer can backtrace which parts of an input affect a specific branch condition. Given this semantic insight, fuzzers can concentrate their efforts on mutating input parts that are relevant to overcome specific constraints. Using symbolic or concolic execution, fuzzers can *compute* values that are required to solve conditions or integrity checks or, more generally speaking, exercise all paths in a given program. While heavyweight feedback-driven fuzzers have proven effective in generating well-structured inputs, they suffer from new challenges and limitations. The main limitation is that they are relatively slow, fail to scale to large target programs, and require runtime environments that specify, e. g., side effects of library functions [4]. Furthermore, complex constraints imposed by cryptographic primitives such as hash functions or signatures cannot be solved due to their computational complexity.

In summary, we find that current approaches to generating complex, highly structured input rely either on a grammar provided by a human expert, which is effective but costly, or approximate the grammar (less costly but less effective). Note that neither of them can handle mutations of complex data. Other methods use heavyweight techniques that do not require a grammar but are inefficient and do not scale. None of the state-of-the-art approaches can leverage existing domain knowledge in an automated and effective manner.

3 Design

In this work, we take an orthogonal approach by changing the perspective of fuzzing: We propose to focus on the *generator* applications that produce the input to the target program under test. Our approach uses a simple yet powerful idea: instead of directly mutating the input to create a new test case, we mutate the generator application and use its output as a fuzzing test case for the target under test. This way, we can (implicitly) leverage domain knowledge and overcome complex constraints *without* suffering from the shortcomings of heavyweight techniques or manual approaches.

Our observation is that generator programs generally produce well-formed outputs, which is indispensable to ensure interoperability. By selectively injecting faults into these generator programs, they produce *almost* well-formed outputs, i. e., they may violate the specifications in subtle ways. This allows us to produce high-quality test cases for the respective consumer program. For example, suppose that we inject faults into instructions that manipulate (i. e., read or write) partially processed data, which will subsequently be cryptographically signed. In this case, we can produce *mutated but valid* inputs with respect to the enveloping signature. Crucially, these inputs are not discarded early during signature validation, but reach deeper program logic in the consumer.

Figure 1 presents a high-level overview of our approach and shows how the individual components of our design interact. On a high level, we want the generator application to produce diverse test cases that we can supply to our actual fuzzing target, the consumer. To this end, the fuzzing scheduler selects ❶ one or multiple mutation(s) for the generator application (e. g., changing a value stored by a `mov` instruction) and, if needed, a seed file processed by the generator. Next, ❷ the selected mutation(s) are applied to the generator. The mutated generator now processes the input ❸ and produces ❹ a slightly mutated output. Finally, the scheduler passes ❺ the generated output to the consumer, and coverage feedback is collected ❻. If the test case triggers interesting behavior (new coverage), the pair of mutated generator program and input file is enqueued for further mutation in subsequent rounds.

3.1 Generator

In essence, the generator can be considered a seed generator for producing inputs tailored to the fuzzing target, the consumer. While common fuzzing approaches mutate inputs on the fly through bit-level mutations, we mutate inputs indirectly by injecting faults into the generator program. More precisely, we identify and mutate *data operations* the generator uses to produce its output.

Generator Requirements. To facilitate our approach, we require a program that generates outputs that match the input format the fuzzing target expects. Most generator applications, such as image converters, require files that can be converted

Figure 1: High-level overview of our approach for a generator and a consumer program. Noteworthy, the generator may produce outputs (❹) that slightly violate the specification.

to the target format. Thus, in contrast to traditional fuzzing approaches, the initial seed files are inputs for the generator instead of the consumer. Based on the type of generator, seed inputs may not be required. This applies, for instance, to generators of cryptographic keys.

Data operations. One core challenge for mutating the generator is to identify and mutate *data operations*. We consider any instruction reading from or writing to memory as a data operation. The underlying insight is that output generated by a program is stored in memory at one point or another in almost all cases, especially for the type of complex data formats that we target in this work. For a particular input to the program, we say it *covers* the data operation when it executes the respective operation (an essential requirement for a mutation to have any effect). Moreover, we say that the data operation is *mutated* when the underlying data is modified with respect to the original program.

To reliably identify all data operations within a program, we design a compiler pass that allows us to instrument all *load* and *store* operations. These operations are ubiquitous in programs, resulting in many potentially interesting operations. Consequently, we cannot instrument every instance. Instead, we must identify a subset of data operations that gives us maximum control over the processed data while minimizing adverse runtime effects, such as crashes. Besides their sheer number, several factors determine an instruction's relevance:

- *Impact*: Does this particular instruction modify relevant data, i. e., data that has an observable impact on the output? Modifying instructions unrelated to the generated data has no benefit towards the goal of producing interesting input files for our fuzzing target.
- *Type*: Does the modified data represent a value that is likely to cause a crash of the generator if it is mutated, e. g., because it is a function pointer? Modifying pointers instead of actual values is prone to crash the application instead of producing interesting values.
- *Data-flow Dependency*: Does the data depend on a value that has already been modified by earlier instructions? Intuitively, it is undesirable to modify the same data

more than once since the initial modification can already produce any possible value; in the worst case, the second change can cancel the changes of the first. This does not apply for partial dependencies.

To address these challenges, we design our compiler pass to only instrument instructions loading or storing value types and avoid pointer types. Additionally, we use the data-flow information available to the compiler to instrument only the first instance of a data value being modified. Unfortunately, static analysis during compile-time falls short in determining whether a particular instruction will have a significant impact (if at all) on the application's output. Consequently, before starting the actual fuzzing process, our design budgets for a lightweight identification and pruning phase during runtime, a process we describe in the following.

Instrumentation Site Pruning & Impact Analysis. We have observed that allowing the fuzzer to inject faults at all potential instrumentation sites leads to many ineffective mutations. For example, a given input file exercises only a specific path in the (potentially mutated) generator's code, which commonly only includes a fraction of all instrumentation sites. We call these sites *dead* w.r.t. a specific input and (mutated) version of the generator. Even *alive* (i. e., not dead) instrumentation sites may have no actual *impact*. This means that, for a given combination of input and generator version, injecting a fault into this instrumentation site will not lead to any coverage change in the consumer.

Intuitively, to maximize the number of effective fault injections, we want to avoid mutating dead instrumentation sites and ones without a visible impact. Consequently, the goal of our pruning and impact analysis phase is twofold: For each combination of seed input and (potentially) mutated generator, we first aim to identify and remove dead instrumentation sites and then analyze the impact of the remaining ones. To this end, we trace the execution and keep only the alive instrumentation sites. Additionally, to analyze the impact of alive instrumentation sites, we observe whether injecting a fault into the underlying data operation yields a different code coverage in the consumer. This stage results in a list of alive and impactful instrumentation sites per seed file and (potentially) mutated generator application. Furthermore, we record additional information, such as the number of times an instrumentation site is exercised, for later use, e. g., when generating mutations.

Mutations. To generate slightly corrupt inputs, we mutate the generator: We randomly select an input for the generator and one or more instrumentation sites. We then apply bit-level mutations to the data values processed at the respective instrumentation sites. As these sites can be visited multiple times during program execution, e. g., in loops, we can either always apply the same mutation or opt for more fine-granular control in the form of independent mutations that differ for each visit. For the latter, we need *a priori* knowledge of *how*

often a particular input will visit an instrumentation site. This information is conveniently available from our *pruning & impact analysis* pass. This allows us to specify a list of mutations for each instrumentation site. Each visit of the mutation site consumes one of its mutations, until all mutations for a given instrumentation site are consumed. In the rare case a mutation modifies control flow, such that an instrumentation site is visited more often than observed during the impact analysis, we can not apply any further mutation but return the unmodified data value.

3.2 Consumer

The generator's counterpart is the consumer: As the target of our fuzzing campaign, it consumes the inputs generated by the generator. Similar to typical fuzzing targets, we impose no specific restrictions or limitations on the consumer. Since we use coverage feedback to guide our mutations in the generator, the consumer must provide an interface to retrieve coverage information (via instrumenting the source code or from emulation).

3.3 Scheduler

The last component is the scheduler, which orchestrates the interaction of the generator and the consumer. It governs the fuzzing campaign, and its main task is to organize the fuzzing loop. The scheduler contains the following components:

Queue. The scheduler maintains a queue containing *queue entries*. Each entry consists of the seed input passed to the generator (if any) and all mutations which have been applied to the generator. Each such queue entry represents a single test case. In traditional fuzzing, such a test case would be represented as a single file.

Phases. The main fuzzing loop is split into multiple *phases*, see Algorithm 1. Depending on the phase type, the steps within a phase are performed exactly once for each new queue entry (*calibration phase*) or several times during fuzzing. Upon launching a fuzzing campaign, all seed files are added to the queue and thus calibrated.

1. **Calibration Phase**: We pass the input to the (potentially mutated) generator and record the instrumentation sites visited during execution (*instrumentation site pruning*). For each instrumentation site of the target, we further assess its impact on the coverage in the consumer (*impact analysis*). This information, alongside the input and mutations, is stored within the queue entry. Importantly, this phase is part of the regular fuzzing iteration.

During the main fuzzing loop, we then repeatedly pick one queue entry and select one of the following phases:

2. **Add Phase**: We pick multiple instrumentation sites and apply mutations. We prefer sites that have previously successfully yielded new code coverage in the consumer. This phase *adds* new mutations to the queue entry, extending the instances of *mutated* data operations.

3. **Mutate Phase**: We apply a fixed number of mutations to all mutated data operations of the selected queue entry. In contrast to the add-phase, this phase does not add any new data operations but mutates existing ones.

4. **Combine Phase**: For each mutated data operation, we inspect whether *other* queue entries have mutated it as well. If so, we try their mutations. This is similar to splicing, known from traditional fuzzing, and allows to benefit from mutations that have already proved to affect coverage.

If we observe new code coverage during the main fuzzing loop, we need to execute the calibration phase for the new queue entry, which is created to represent the combination of input and mutations that yielded the new coverage.

Our queue entry selection algorithm is similar to the one used by AFL. Both use the concept of novelty search, i. e., we keep inputs based on whether they yielded new code coverage. Furthermore, we apply a similar favoring scheme that prioritizes a minimal set of inputs covering a maximum of the code in the consumer. The main difference to AFL is that we prefer queue entries containing rarely observed data operations; AFL has no similar concept given that the method does not observe data operations at all.

3.4 Combined Fuzzing

Compared to traditional fuzzer designs, our input generation method is slower: instead of flipping a byte, the generator program is mutated and executed to produce an input. Furthermore, AFL-based fuzzers are capable of splicing or splitting inputs—operations that a generator typically does not expose. To compensate for these missing operations and the performance impact, our approach can be used in tandem with a traditional fuzzer such as AFL++. This approach is similar to fuzzers such as QSYM [5], SYMCC [6], or DRILLER [14], which use AFL for regular fuzzing and augment it by providing new inputs that solve fuzzing roadblocks which common mutations cannot address. In the same vein, we propose an approach focusing on a generator application to produce interesting inputs that unlock new, deeper state space for the traditional fuzzer.

4 Implementation

We implement our design in a prototype called FUZZ-TRUCTION that consists of about 14, 000 lines of Rust code. Next we discuss implementation aspects.

Algorithm 1: Simplified algorithm representing our approach. Before the main fuzzing loop starts, all seed inputs are calibrated. This includes the instrumentation site pruning and analysis of the data operations' impact w.r.t. the target's (i. e.,consumer's) coverage. Finally, the fuzzing loop is executed until it is stopped.

```
1  Input: Seeds
2  Q ← create_queue()
3  forall seed ∈ Seeds do
4  |    Q ← Q ∪ calibration_phase(seed)
5  while True do
6  |    entry ← select_next(Q)
7  |    choice ∈_Random {1..3}
8  |    switch choice do
9  |    |    case 1 do
10 |    |    |    result ← add_phase(entry)
11 |    |    case 2 do
12 |    |    |    result ← mutate_phase(entry)
13 |    |    case 3 do
14 |    |    |    result ← combine_phase(entry, Q)
15 |    if crash_or_new_coverage(Q, result) then
16 |    |    Q ← Q ∪ calibration_phase(result)
```

Generator.   To instrument the generator, we develop a compiler pass for LLVM which identifies all data operations and prepares them for mutation. We use an experimental LLVM feature called *stack maps* to create an instrumentation site for each data operation. In essence, this means that the compiler records the location of instruction arguments during runtime, e. g., in which register an argument is passed to a *store instruction* (stack map record). In conjunction with another LLVM feature called *patch points*, which places padding (in the form of `nop` instructions) at the position of a stack map record, this allows us to inject arbitrary code that mutates the operands of each data operation. We Just-In-Time (JIT) compile dynamically generated *JIT stubs* into the padding provided by patch points. To keep the required padding size small and predictable, we opt for trampolines that call into code that we allocate in a separate, executable section.

As *stack maps* and *patch points* are an experimental feature of LLVM, they do not correctly handle all corner cases (such as vector operations). We developed patches for LLVM 11.0.1 and LLVM 12.0.1 to introduce the missing support [1]. Beyond implementing the instrumentation pass, we insert a runtime component that implements a forkserver to quickly execute multiple inputs and develop the JIT compiler to apply, remove, and generate *JIT stubs*. We use these stubs to implement two types of functionality: (1) The tracing required for the instrumentation site pruning and impact analysis phase, and

---

[1] We release the patches alongside our code.

(2) the mutation of data operations themselves.

Instrumentation Site Pruning and Impact Analysis.   Using the JIT compiler, we facilitate the analysis phase by injecting a callback to a custom logging function into each instrumentation site. Based on these callbacks, we determine which instrumentation sites are alive. As an additional piece of metadata, we also count how often they are visited (i. e., executed) for a specific input. These execution counts allow the fuzzing loop of the scheduler to determine how many data operations can be mutated at each instrumentation site.

Mutations.   The second type of JIT stub implements the application of mutations to data operations. In FUZZTRUCTION, we implement a mutation on a data operation by XORing a bit mask into its data operand. For load operations the instrumentation site is placed after the operation, while for stores the site is placed before the data operation. Figure 2 shows how a store operation is mutated based on a provided list of bit masks (*mutation masks*). Beyond XORing, we could implement other operations to mutate the data, e. g., setting it to specific value or incrementing/decrementing it. This is analogous to mutations used in traditional fuzzing. However, these mutators empirically often yield only few coverage, thus we do not implement them.

We use the number of observed executions of each instrumentation site we collected during tracing as a hint towards how many mutations (i. e., bit masks) can be inserted for each instrumentation site. We also keep track of the instrumentation sites for which mutations yielded new code coverage in the consumer, prioritizing them while subsequently picking instrumentation sites to mutate.

Generator Stability.   Given that we mutate the generator application, and despite the analysis passes described in Section 3, we risk modifying the generator in such a way that input causes it to crash instead of producing an output. If we detect such a case, we remove the offending instrumentation site from the set of sites that are picked for mutation. While this may appear conservative (as there can exist other mutations that would not crash), we empirically found it is very unlikely that a crashing instrumentation site recovers. Similarly, we detect stalls in the generator by setting a timeout of a few milliseconds and handle them in the same way as generator crashes. Furthermore, to prevent generators from negatively affecting the host filesystem, we jail them such that they cannot modify files beyond their output.

Consumer.   To collect code coverage from the consumer, we use the AFL-compatible fork server interface. If source code is available, we apply AFL's compile-time coverage instrumentation. Otherwise, we can fall back to QEMU user mode instrumentation for binary-only programs.

To avoid unnecessary executions of the consumer, we hash the inputs, i. e., the data produced by the generator, and only execute those not tested before.
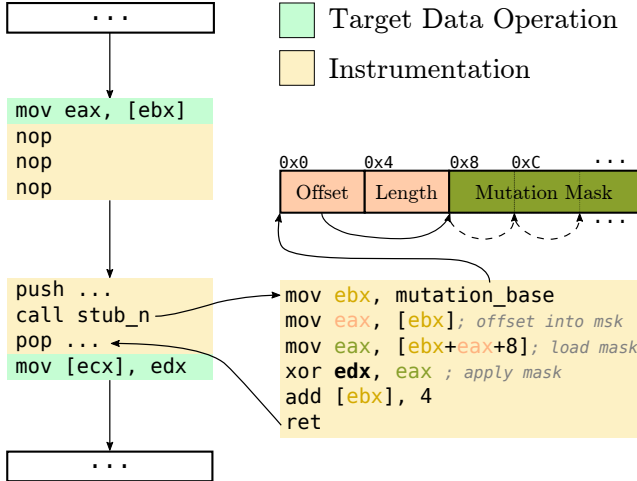
Figure 2: Technical perspective of how JIT stubs are used to perform a mutation on different types of data operations. First, a load operation is left unmodified. In the second instrumentation site, an operand of a store operation is modified before it is commited to memory. The example assembly code is simplified in that it assumes a 32-bit process and omits bounds checks. The stored data value is modified by XORing a different bit mask into the edx register each time before the mov instruction is executed.

Scheduler. In addition to the compiler passes instrumenting generator and consumer, we implement a scheduler that orchestrates the whole fuzzing process and communication between generator and consumer. Furthermore, we integrated support for using AFL++ to apply simple bit level mutations to inputs found by FUZZTRUCTION. Consequently, we can spend more time on unlocking new program compartments while leaving the task of discovering these to a fuzzer that is suited to achieve high test case throughput.

5 Evaluation

In this section, we evaluate our prototype FUZZTRUCTION to gain deeper insights into where our approach applies and how our prototype performs compared to state-of-the-art fuzzers.

5.1 Setup

We first describe our experimental setup for the evaluation including the hardware environment, the fuzzers we are evaluating against, and the target programs.

Hardware Environment. We use the same hardware configuration for all experiments: An Intel Xeon Gold 5320 CPU @ 2.20GHz (52 physical cores), 256 GB of RAM, and SSD memory as backing storage.

Fuzzers. We evaluate the following five fuzzing methods. As a baseline, we use FUZZTRUCTION-NOAFL and AFL++, the two components used by our approach. We evaluate our

prototype implementation of FUZZTRUCTION against two other methods, SYMCC [6] and WEIZZ [30], representing the state of the art regarding heavyweight program analysis and approximation of input structure. None of the fuzzers require any precomputation.

1) FUZZTRUCTION-NOAFL. This stand-alone variant of FUZZTRUCTION is not paired with AFL++, but instead relies solely on the inputs produced by the generator application. As a consequence, it has no access to traditional mutations, especially the freedom of splicing and splitting inputs.

2) AFL++. The second baseline is AFL++ [1] (version 4.00c): An approach representing the traditional byte-level mutation-oriented fuzzers. Being constantly developed and improved, AFL++ represents state-of-the-art greybox fuzzers and is used by many specialized fuzzers, such as SYMCC.

3) FUZZTRUCTION. This method represents our approach presented in this paper. We augment AFL++ with the idea of mutating generator programs, basically combining the two techniques mentioned above.

4) SYMCC. We choose SYMCC [6] (commit `07c8895`) as a representative method for heavyweight program analysis-based approaches, here symbolic execution. SYMCC uses compiler-based instrumentation to leverage LLVM optimization passes to make constraint extraction more feasible. Similar to FUZZTRUCTION, it is paired with AFL (in our experiments with AFL++) and used to solve constraints that AFL cannot solve. To the best of our knowledge, SYMCC is the only state-of-the-art concolic fuzzer that does not require any sort of warm-up to, e. g., collect constraints, and therefore is comparable to our approach. As SYMCC requires a description of library functions, such that constraints can be carried across the library calls, e. g., `libc`, Poeplau and Francillon have annotated several functions. Unfortunately, others, such as `open64` or `pread64` are missing. Without descriptions, SYMCC will not work for a target calling these. We manually add missing annotations, such that SYMCC works for the targets in our evaluation.

5) WEIZZ. WEIZZ [30] is a fuzzer that approximates, during run-time, the input structure based on input-to-state correspondence. It shares FUZZTRUCTION's idea of creating complex structured inputs and uses a REDQUEEN-like [4] approach to profit from input-to-state correspondence, which has been successfully used to overcome roadblocks such as checksums [4]. We use WEIZZ (commit `c9cbeef`) as provided for our evaluation: Contrary to SYMCC and FUZZTRUCTION, WEIZZ is tightly coupled to AFL and a *binary-only* approach using AFL's QEMU mode instead of source instrumentation. This gives it a slight disadvantage compared to the other fuzzers, which have access to the faster compile-time instrumentation. However, we still include it in our evaluation as it is, to the best of our knowledge, the most powerful approach in the domain of grammar inference *paired* with techniques to overcome roadblocks.

Target Applications. We select ten different applications based on several factors to determine the effectiveness and efficiency of our approach. We mark applications using cryptographic primitives with a lock symbol, 🔒. The symbol (🔒) indicates that a program uses these primitives only optionally. Overall, our goal was to select different groups of applications that process three different types of input formats:

- **Loosely Structured Formats** (`objdump`, `readelf`): These formats do not employ complex constraints and are chunk-based. The targets of this group are already well-tested by traditional fuzzers which employ bit-level mutations (AFL++) or inference for chunk based formats (WEIZZ).

- **Complex Formats** (`pngtopng`, `unzip`, `7zip`(🔒), and `pdftotext`(🔒)): Such programs usually exhibit an input structure that is challenging to fuzz, e.g., because of transformations, such as compression or checksums. As a result, more sophisticated approaches than traditional byte level-oriented mutations, e.g., symbolic execution or structure inference paired with input-to-state correspondence (WEIZZ), are necessary to achieve high coverage. We include targets using cryptography optionally, as fuzzers can create inputs exercising deep program states without using cryptography, e.g., because only some chunks of the inputs are encrypted.

- **Cryptographic Formats** (OpenSSL's `dsa`🔒 and `rsa`🔒, and Mozilla NSS' `vfychain`🔒): These applications elude state-of-the-art fuzzers, primarily due to using cryptographic primitives.

All applications used during our evaluation are described in more detail in Appendix B. As commonly done in fuzzer evaluations, we use code coverage as a proxy of how well a fuzzer performs. To discuss whether this reflects the fuzzers' ability to find bugs, we refer the interested reader to Böhme et al. [36]. We use no sanitizers as they do not exhibit new code coverage (to triage bugs, we use `Valgrind` as described in Section 5.3). Noteworthy, SYMCC runs into segmentation faults for `7zip` and fails to build `vfychain`🔒 because it does not support some vectorized instructions (i.e., an assert is triggered during compilation). We exclude it from these targets as a consequence.

Target Preparation. All targets under test were prepared according to the respective fuzzers' needs. Since AFL++, FUZZTRUCTION, and SYMCC rely on the AFL++'s compiler pass instrumentation, all targets (for FUZZTRUCTION, only the consumers) were compiled via `afl-clang-fast` in version 4.00c. In addition to the default settings, we also set the following flags [1, 37]:

- `AFL_LLVM_LAF_SPLIT_SWITCHES=1`
- `AFL_LLVM_LAF_SPLIT_COMPARES=1`
- `AFL_LLVM_LAF_TRANSFORM_COMPARES=1`

Furthermore, since SYMCC is incompatible with the default, collision-free instrumentation schema used by AFL++, we set `AFL_LLVM_INSTRUMENT` to `CLASSIC` for SYMCC, such that the injected instrumentation remains backward compatible with plain AFL. We use the SYMCC compiler wrapper to inject the constraint recording logic. For WEIZZ, we use the same uninstrumented binary as for the coverage computation. Finally, since we need generator applications for FUZZTRUCTION to mutate, we compile for each target (consumer) application a generator application with our custom LLVM compiler pass.

Seeds. To create seeds for each target, we use existing seed sets for the generator of each respective consumer or create basic ones from scratch. A description of each seed input can be found in Appendix C. In some cases, such as `genrsa`, we do not provide any seed corpus to the generator since the application does not consume any input. To ensure fairness, the seed file sets for the consumer are generated by executing the generator on all generator seed files and using the resulting files as seed file sets for AFL++, SYMCC, and WEIZZ. Additionally, in case the generator and consumer process the same kind of format, we also provide the unprocessed generator inputs to the other fuzzers.

Coverage computation. We use coverage as a metric to evaluate fuzzer performance. For this, we use the `drcov` submodule of the DYNAMORIO [38] tracing framework. This allows us to retrieve execution traces (i.e., start addresses of all executed basic blocks) for a given input on an uninstrumented binary. Running the inputs from all fuzzers on the same uninstrumented binaries ensures that coverage numbers reported in this paper are consistent and comparable. To reduce noise generated during tracing, we excluded the following standard libraries from being traced: `libgcc`, `libstdc++`, `libc`, `libpthread`, `libm`, `libdl`, and `ld-linux`.

Table 1: Fuzzing roadblocks in our targets alongside their generators. Some targets heavily rely on checksums or crypto. Several targets, such as `7zip`, can work with unencrypted and encrypted data; (✓) indicates that these targets use cryptographic as an opt-in. More details on the applications can be found in Appendix 5.

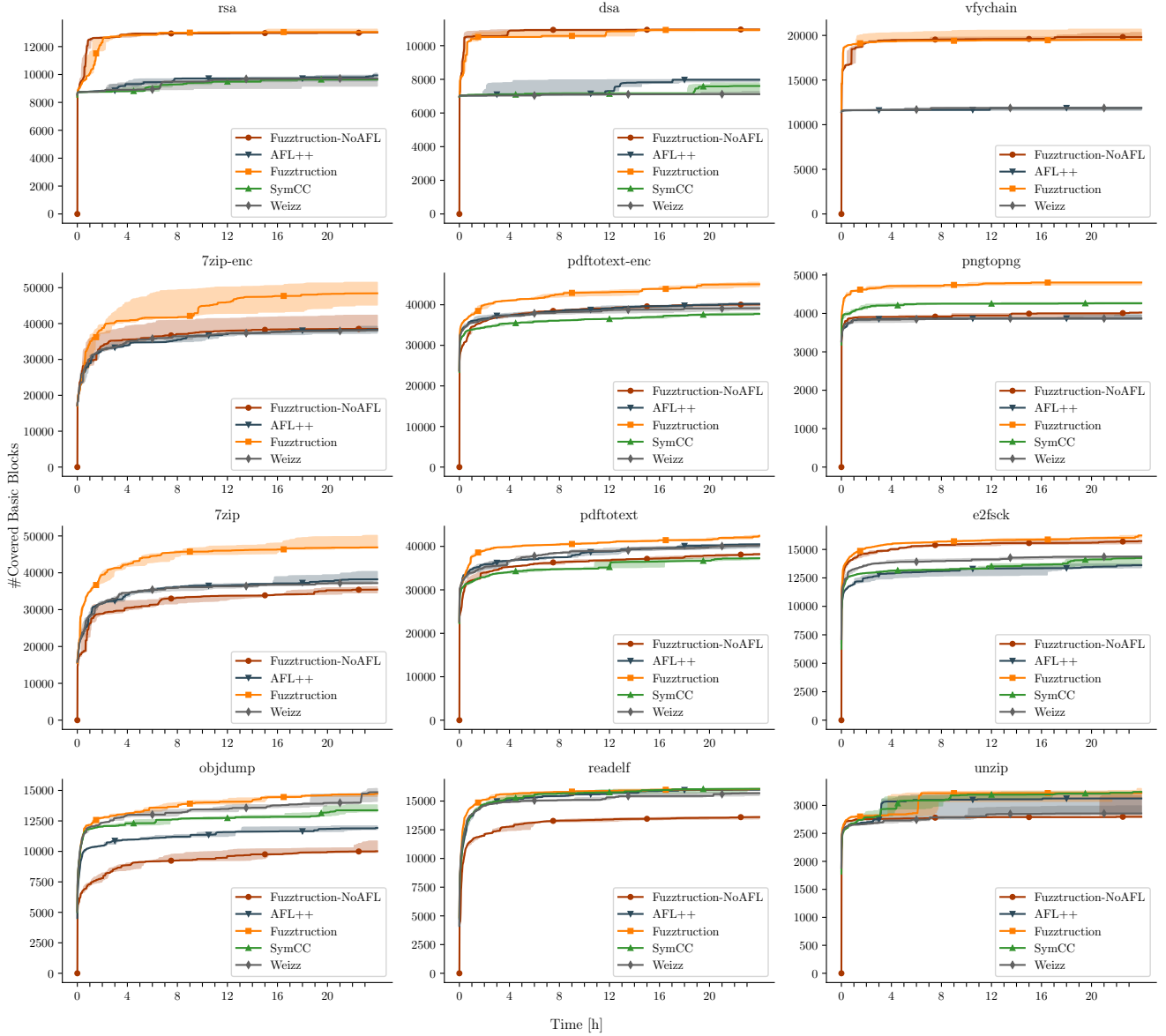| Target | Roadblocks | | Generator for FT |
	Checksums	Crypto	
`rsa`🔒	✓	✓	`genrsa`🔒
`dsa`🔒	✓	✓	`gendsa`🔒
`vfychain`🔒	✓	✓	`sign`🔒
`7zip`(🔒)	✓	(✓)	`7zip`, `7zip`🔒
`pdftotext`(🔒)	✓	(✓)	`pdfseparate`, `qpdf`🔒
`unzip`(🔒)	✓	(✓)	`zip`
`pngtopng`	✓	✗	`pngtopng`
`e2fsck`	✓	✗	`mke2fs`
`readelf`	✗	✗	`objcopy`
`objdump`	✗	✗	`objcopy`

Figure 3: The coverage (in basic blocks) produced by various tools over five 24h runs on different targets. Displayed are the median and the 60% intervals. SYMCC crashes on `7zip` and `vfychain`🔒; thus we have excluded it from these targets.

5.2 Coverage Experiments

To evaluate the effectiveness of our approach, we compare FUZZTRUCTION with AFL++, SYMCC, and WEIZZ using the ten application (pairs) displayed in Table 1. For each of the ten targets, we repeatedly run each fuzzer five times, for 24 hours on 52 cores (if a fuzzer has multiple baselines, e. g., FUZZTRUCTION, we fairly split the cores, giving each 26).

The results of these experiments are shown in Figure 3. At first, we take a look at the overall results and how our approach performs in general. In the majority of all cases, FUZZTRUCTION covers the most basic blocks. In two cases,

`readelf` and `unzip`, most fuzzers perform equally, while FUZZTRUCTION is still on par with the other candidates. Only in a single case, for `objdump`, FUZZTRUCTION is outperformed by WEIZZ by a small margin.

> FUZZTRUCTION outperforms state-of-the-art methods in terms of overall code coverage.

In the following, we analyze the results in more detail for each of the application groups described in *Target Applications* in Section 5.1. In particular, we consider two dimensions: First, we inspect FUZZTRUCTION's individual

baselines, AFL++ and FUZZTRUCTION-NOAFL, and analyze *if* and *how* they synergize. Second, we compare how FUZZTRUCTION performs relative to SYMCC and WEIZZ, which represent the state-of-the-art of more traditional fuzzing approaches.

Loosely Structured Formats. This group, represented by `readelf` and `objdump`, represents the baseline of targets which general purpose fuzzers are commonly evaluated against. FUZZTRUCTION-NOAFL, as an isolated baseline of FUZZTRUCTION, fares worse than its other baseline AFL++. This can intuitively be expected: Traditional fuzzers such as AFL++ use simple bit mutations which are effective in exploring common, chunk-based binary file formats as they feature a high throughput in their input generation. This is in contrast to the fault injection-based mutation introduced in this work, where generating an input requires the generator application to be run to produce a new input. Measuring the actual executions per second for all targets, FUZZTRUCTION-NOAFL performs by a factor of 3.2 slower than AFL++. Beyond throughput, there is a second reason for the difference in coverage found: FUZZTRUCTION-NOAFL has no notion of *splicing*, which is particularly useful for chunk-based formats—such as ELF files processed by `objdump`.

Apart from the rather low individual performance of FUZZTRUCTION-NOAFL, we find that combining the two baselines (FUZZTRUCTION) yields synergies for `objdump`: FUZZTRUCTION-NOAFL uniquely covers functions that handle complex format parts such as compressed ELF sections or ones in formats that are entirely different from the format of the provided seed files (e. g., Common Object File Format (COFF)). This way, FUZZTRUCTION-NOAFL provides high-value inputs to AFL++, such that FUZZTRUCTION profits from both.

Compared to WEIZZ and SYMCC, we observe a similarly intuitive overall picture: As the other fuzzers are also optimized for testing chunk-based binary targets, the other fuzzers perform comparatively well, where WEIZZ outperforms FUZZTRUCTION for `objcopy`, whereas all fuzzers perform similarly for `readelf`.

> FUZZTRUCTION's fault injection-based input generation falls short on traditional fuzzing targets, but produces inputs unlocking new coverage in several cases.

Complex Formats. The next group contains `pngtopng`, `unzip`, `e2fsck`, `7zip`[🔒], and `pdftotext`[🔒] in the middle rows of Figure 3. These programs feature more sophisticated challenges for fuzzers, namely checksums and transformations such as (chunk-wise) encryption or compression. Regarding the interplay between the baselines of FUZZTRUCTION, we see that in comparison to the previous set of applications, FUZZTRUCTION-NOAFL is closer in individual performance to AFL++ than before. As FUZZTRUCTION,

the combination of the two, performs better than its respective baselines, we can see that FUZZTRUCTION-NOAFL contributes even more high-quality inputs for complex formats than for loosely structured formats.

Notably, FUZZTRUCTION-NOAFL performs worse than AFL++ on targets when they do *not* use cryptographic primitives (here, password-based encryption of input files). In contrast, if the input files are encrypted, FUZZTRUCTION-NOAFL can show its strengths and performs about as well as AFL++. This implies that our approach is capable of generating interesting encrypted input files. To verify that notion, we inspect the uniquely covered functions for all fuzzers and find that FUZZTRUCTION is the only fuzzer to cover various encryption-related functions. These inputs, in turn, unlock AFL++'s byte-level mutations within FUZZTRUCTION to find new coverage, showcasing their synergy effects. Interestingly, FUZZTRUCTION's coverage intervals for `7zip`[🔒] significantly differ from the median after 24 hours, indicating it has not yet converged. Anecdotally, when running FUZZTRUCTION for 72 hours on `7zip`[🔒], we indeed find it still uncovered new coverage after 24 hours. Another interesting target is `e2fsck`, where FUZZTRUCTION and FUZZTRUCTION-NOAFL perform nearly equally, indicating that AFL++ is unable to contribute any meaningful test cases. Looking at the test cases produced by the generator, `mke2fs`, we find that one reason for our approaches' good performance is the fact that our mutations managed to generate different file system versions, such as `ext2`, `ext3` or `ext4`.

Comparing the coverage produced by FUZZTRUCTION to WEIZZ and SYMCC, the synergies between FUZZTRUCTION's components also become more clearly visible. FUZZTRUCTION outperforms WEIZZ and SYMCC by a more significant margin than for the last set of targets.

> The synergy effects of combining FUZZTRUCTION-NOAFL and AFL++ are clearly visible for targets that impose complex transformations, such as compression or chunk-wise encryption onto their input.

Cryptographic Formats. The last group entails the targets `rsa`[🔒], `dsa`[🔒], and `vfychain`[🔒] (top row in Figure 3). These targets are characterized by the fact that they implement complex cryptographic primitives such as asymmetric cryptography or operations, such as signing, which is typically applied to certificates. These values are complex since they have an inner structure defined by the underlying mathematical primitives, which is likely voided if mutated.

For all targets in this group, FUZZTRUCTION-NOAFL and FUZZTRUCTION perform equally well. One interesting insight is that AFL++ does not meaningfully contribute to FUZZTRUCTION's coverage and also does not benefit from the seeds generated by our approach. This is because it instantaneously breaks the inner structure of these inputs by applying its bit-oriented mutations. By more closely inspecting the
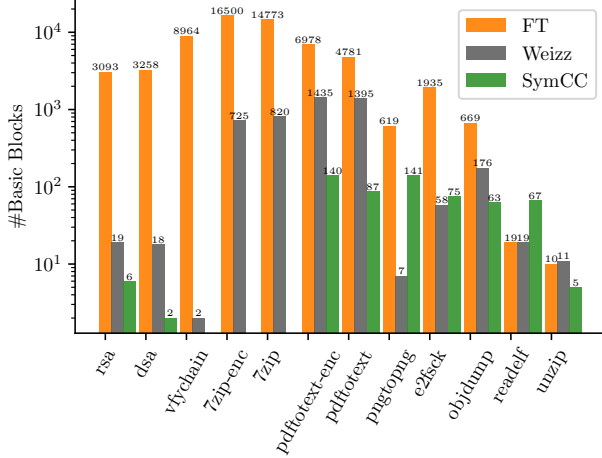
Figure 4: Logarithmic plot showing the number of basic blocks *exclusively* found by FUZZTRUCTION (FT), WEIZZ, or SYMCC. SYMCC does not support vfychain🔒 and 7zip (🔒).

Target	Best Competitor	p-value	$\hat{A}_{12}$ effect size
rsa🔒	AFL ++	**< 0.05**	+L (1.00)
dsa🔒	AFL ++	**< 0.05**	+L (1.00)
vfychain🔒	AFL ++	**< 0.05**	+L (1.00)
7zip🔒	AFL ++	**< 0.05**	+L (1.00)
pdftotext🔒	AFL ++	**< 0.05**	+L (1.00)
pngtopng	SYMCC	**< 0.05**	+L (1.00)
7zip	AFL ++	**< 0.05**	+L (1.00)
pdftotext	AFL ++	**< 0.05**	+L (1.00)
e2fsck	WEIZZ	**< 0.05**	+L (1.00)
objdump	WEIZZ	1.0000	(0.48)
readelf	SYMCC	0.8413	(0.44)
unzip	SYMCC	0.6905	-S (0.40)

unique functions covered by FUZZTRUCTION in case of rsa🔒, we find that FUZZTRUCTION uniquely covers 298 functions related to implementations of different encryption and hashing algorithms, like sha256, sha512, AES, and IDEA; it even triggers the use of different cipher modes. This is particularly noteworthy as the generator does not consume any seed input for this target. Consequently, the variety in outputs resulting from our approach succeeds in generating high-quality inputs for complex structures, even without seed inputs.

Other fuzzers struggle with this type of targets: Since the mathematics underpinning public key cryptography are designed as one-way operations, it is impossible for symbolic execution to generate valid key pairs out of thin air. AFL++ and WEIZZ both void the cryptographic primitives due to the bit-level mutations and WEIZZ does not benefit from its approach since it can not infer valid signatures or encryption. As a result, FUZZTRUCTION significantly outperforms other fuzzers for cryptographic applications.

> Our approach is excellent for fuzzing cryptographic applications and represents the only way of fuzzing such applications without relying on manual harnessing.

Exclusively Covered Basic Blocks. To further quantify the difference between the state-of-the-art techniques tackling the creation of complex input formats, we analyze the basic blocks found *exclusively* by a single fuzzer from FUZZTRUCTION, WEIZZ, and SYMCC. As visible in Figure 4, FUZZTRUCTION finds significantly more basic blocks (that are not found by any other fuzzer) for all targets but readelf and unzip. This indicates that FUZZTRUCTION covers code the other two fuzzers failed to explore.

Statistical significance. Following Klees et al.' [39] as well as Arcuri's and Briand's [40, 41] recommendations, we verify whether the observed differences are statistically significant. To do so, we use the two-sided non-parametric Mann-Whitney-U test [42]. Additionally, we measure effect sizes to quantify the improvement. To this end, we conduct the non-parametric Vargha and Delaney $\hat{A}_{12}$ test [43, 44]. It measures the probability that running FUZZTRUCTION yields higher coverage values than its best performing competitor (AFL++, SYMCC, or WEIZZ): If both algorithms are equivalent, $\hat{A}_{12} = 0.5$; if, for instance, $\hat{A}_{12} = 0.9$, 90% of the time, FUZZTRUCTION achieves better coverage results than the fuzzer we compared it to. Based on Vargha and Delaney's guidelines [43], we consider an effect size $\hat{A}_{12} > 0.56$ as small, > 0.64 as medium, and > 0.71 as large. For the sake of simplicity, we only report the difference to the *best-performing* competitor (chosen by median coverage).

The results in Table 2 show that FUZZTRUCTION is significantly better for all but three targets, i.e., the difference in the number of found basic blocks is statistically significant (indicated by p-value < 0.05) and the effect size is large ($\hat{A}_{12} > 0.71$). Unsurprisingly, this does not hold for objdump and readelf, targets where traditional fuzzers already perform well. Here, WEIZZ and SYMCC are slightly better (in the median coverage), however, according to Mann-Whitney-U, there is no statistical difference for objdump and readelf. Additionally, in both cases, the effect size does not meet the bar suggested by Vargha and Delaney [43]. Only for the third target, unzip, a small negative effect size is visible, however, it is statistically insignificant (p-value > 0.05).

Overall, these results support our intuition that FUZZTRUCTION finds significantly more coverage on most targets. For the other targets, we could not find a significant difference

Table 3: Unique crashes found by different fuzzers. All crashes have been bucketed by hashing the last three functions of `Valgrind`'s backtrace (or of all backtraces if `Valgrind` reports multiple, e. g., for adjacent allocations of the memory location accessed by an invalid write). We mark targets for which a fuzzer failed to run with a dash. Best result is marked in bold.

Target	SYMCC	WEIZZ	AFL++	FT-NoAFL	FT	Total
rsa🔒	0	0	0	0	0	0
dsa🔒	0	0	0	0	0	0
vfychain🔒	-	0	0	**3**	3	3
7zip🔒	-	2	5	4	**86**	90
pdftotext🔒	0	0	0	0	1	1
pngtopng	0	0	0	0	0	0
7zip	-	4	2	1	**54**	56
pdftotext	0	0	0	0	0	0
e2fsck	1	1	1	6	**7**	10
objdump	0	0	0	0	**8**	8
readelf	1	1	0	0	**2**	2
unzip	23	**31**	25	4	23	38
Sum[1]	25	37	31	17	151	261

[1]) For `7zip🔒` and `7zip`, a fuzzer may inadvertently find the same crash twice (once for `7zip🔒` and once for `7zip`). We count such overlapping bugs only once for the sum.

between FUZZTRUCTION and its best competitor.

5.3 Found Bugs

During our evaluation, we found multiple crashes in the programs under test. Since detecting crashes, and therefore bugs, is the overarching goal of a fuzzer, these findings provide an additional proxy for the effectiveness of our approach. To reduce noise in the huge number of produced crashes, we first ran `Valgrind` on all crashing inputs to categorize them according to the *type* of the underlying memory violation, e. g., segmentation fault, invalid read, or invalid write. For `vfychain🔒`, we also consider uninitialized reads, which are considered security relevant by upstream. Alongside this category, `Valgrind` also provides a stack trace of the function causing the crash and context information, such as stack traces of neighboring allocations. We bucket the crashes based on the last three functions in the stack trace(s). This massively deduplicates the number of crashing inputs (as each bucket contains hundreds if not thousands of crashing inputs), however, it still represents no exact mapping to the underlying bugs. Identifying and triaging bugs requires significant manual effort and often the expertise of a domain expert familiar with the program.

Our results are reported in Table 3. As can be seen, FUZZTRUCTION found the most crashes, with a majority of all crashes occurring in `7zip`. As we fuzz two configurations of `7zip` (🔒), one using encryption and one without, the fuzzers can inadvertently find the same crash for both of them. Investigating how many bugs overlap this way, we find this occurs for all fuzzers (WEIZZ: 2 overlapping bugs, AFL++: 2, FUZZTRUCTION-NoAFL: 1, FUZZTRUCTION: 33). Interestingly, FUZZTRUCTION finds 53 bugs unique to `7zip🔒`,

supporting our argument that it is effective in uncovering bugs in programs using cryptographic primitives. Still, for several targets, no fuzzer finds any crash: This is unsurprising, as targets such as OpenSSL's `rsa🔒` and `dsa🔒` have been audited often and exhibit a comparably small attack surface. Others, such as `pngtopng` (i. e., `libpng`) have been well-tested by previous fuzzing campaigns [33]. Interestingly, SYMCC performs as well as AFL++ despite its subpar coverage. Underlining the synergy effects of combining our approach with AFL++, FUZZTRUCTION finds significantly more crashes than FUZZTRUCTION-NoAFL alone. We manually triaged and reported 27 bugs so far in a coordinated way to the developers. At the time of writing, 19 of these reports were acknowledged as valid by the developers. The reported bugs belong to the following targets (# Acknowledged, # Reported): `unzip` (3/3), `readelf` (1/1), `objdump` (4/4), `pdftotext` (1/1), `7zip🔒` (0/8), `e2fsck` (7/7), and `vfychain🔒` (3/3).

> FUZZTRUCTION significantly outperforms current state-of-the-art fuzzers both w.r.t. to coverage and the number of found security-relevant crashes.

6 Discussion

The novel approach we present in this work is suitable for successfully leveraging domain knowledge in generator applications to generate inputs for complex formats that are both structurally correct and adhere to constraints within the structure, such as cryptographic primitives or compression. We provide a critical discussion of the results, limitations, and possible future work in the following.

Threats to Validity. It is critical to assert the validity of conclusions drawn from experiments, especially if they are empirical. We identify three particularly relevant dimensions for our research and outline our assumptions and the steps taken to ensure the experiments are valid.

External Validity. A major risk is whether any conclusions based on the set of target programs tested can be applied to a broader, more general category of targets. Even though assessing untested software is challenging, we have carefully selected a diverse set of targets covering different categories of programs. In particular, we have not only selected targets employing cryptography, for which our approach is designed and thus likely to succeed, but also targets that have been tested regularly by other fuzzers, such as `objdump` or `libpng` (used by `pngtopng`). Beyond our evaluation, we open-source our implementation to allow anyone to evaluate our approach.

Internal Validity. Beyond generality, it is crucial to minimize systematic errors in the evaluation process itself. We repeat our experiments for all targets five times to avoid any such errors. Furthermore, we evaluate the achieved coverage

for all fuzzers on the same uninstrumented binary to enable comparable and consistent coverage measurements. Finally, to avoid selection bias, we use the same set of seeds files for all fuzzers (or even larger sets for competitors of our approach), as outlined in Section 5.1.

Construct Validity. A final threat to validity is whether the evaluation measures what it is supposed to measure. In general, it is difficult to compare the *approach* implemented by a particular fuzzer with another, since it is highly dependent on engineering factors that are orthogonal to the fuzzer's approach itself. To avoid such discrepancies, we ensure that fuzzer configurations use the same baseline, i. e., we combine them with the same version of AFL++. Thus, when observing changes in coverage relative to the performance of AFL++, we can attribute this change to the paired approach, FUZZTRUCTION or SYMCC respectively. This does not hold for WEIZZ, which cannot be configured this way. Note that WEIZZ and AFL++ share some concepts, e. g., AFL++'s cmpcov feature is inspired by WEIZZ [1]. Additionally, WEIZZ outperforms AFL++ on almost all their tested benchmarks [30]. Hence, we believe the comparison to be acceptable w.r.t. to construct validity.

Requirement for a Generator Application. Intuitively, requiring a generator application for our approach seems like a restriction. However, if an application parses a specific data format or protocol, there is typically a complementing program capable of producing such data. Having a data format without programs producing such data would render the existence of the format itself futile.

Multiple Generator Applications. Having multiple generator applications that implement different set of features w.r.t. to the target data format can be beneficial to cover more code within the target program. Intuitively, this is comparable to having a more diverse seed set for traditional fuzzing, where the seeds cover different code locations and serve as a starting point for later mutations. Similarly, having multiple generators implementing the *same* features may help fuzzing, since different implementations may produce different outputs.

Using an Unmodified Generator. Approximating FUZZTRUCTION's approach, we could use AFL++ to provide mutated input to the generator and feed the outputs produced by the generator into the consumer. Other than for FUZZTRUCTION, the generator itself is here left untouched. This has a number of drawbacks: First, a number of applications does not use any input (genrsa⌗, gendsa⌗, mke2fs, or sign⌗). Second, other applications such as 7zip⁽ᵃ⁾ or zip only wrap the input, here in an compressed container. FUZZTRUCTION's approach can modify the wrapper itself, i. e., produce slightly corrupt zip-files, which AFL++ mutated input data is unable to achieve. Third, for applications not affected by these drawbacks, e. g., qpdf⌗ or pngtopng, using AFL++ to produce input still suffers from AFL++'s inability to overcome more complex constraints. For example, qpdf⌗ expects a PDF file

as input: When fuzzed, AFL++ will make many mutations destroying the PDF's format, such that pdftotext is unable to produce interesting outputs that can be fed into the consumer, pdftotext. Ultimately, this shifts the problem of generating valid input for the consumer to the problem of generating valid input for the generator. FUZZTRUCTION, on the other hand, modifies the generator itself to produce valid but slightly incorrect data.

Seeding. In practice, forming a well-rounded seed corpus is crucial for setting up a successful fuzzing campaign. Prior research has also shown that selecting seeds has an impact on fuzzing results [45, 46]. For a fair evaluation, we use simple, uninformed seed files during our experiments (see Table 6 in Appendix C).

Interactive Generator Consumer Execution. While this is not a limitation inherent to our approach, our prototype implementation does not support bidirectional communications, e. g., client-server applications. Although it is generally possible, supporting such scenarios introduces new challenges that demand consideration. For example, it requires a new oracle to determine whether a fuzzing iteration is over or if one of the participants is still processing data and about to send an answer to the other party. Apart from that, we believe our approach could be well suited to fuzz complex protocols such as TLS that cannot be fuzzed by current approaches [47, 48], mainly due to the cryptographic primitives used.

7 Related Work

Our approach opens a new avenue towards overcoming the problem of solving complex constraints. Previous research has proposed several different approaches to tackle different aspects of the same problem.

Hybrid Fuzzers. To address the shortcomings of blind fuzzers and feedback-driven fuzzers in solving complex constraints, several hybrid fuzzers have been developed. These fuzzers commonly employ advanced program analysis techniques to assist the fuzzer. They identify difficult-to-solve constraints and compute an input that passes this constraint. In theory, this unlocks the fuzzer by showing it how to bypass this particular roadblock. Frequently used program analysis techniques are taint tracking [17–19] and concolic/symbolic execution [5, 6, 14, 16]. While these techniques work for constraints imposed by checksums and similar constructs, they fail for more complex constraints as imposed by cryptographic primitives, as also becomes visible in our evaluation.

Moreover, symbolic execution fails to scale to large programs due to the path explosion problem and requires descriptions of the execution environment. Our approach, however, fulfills the same role as these techniques, but without their shortcomings: Instead of trying to extract the domain knowledge needed to solve a particular constraint from the

fuzzing target (using heavy-weight analysis techniques), we use a *second* program to generate such data. While not directed towards solving one particular constraint, our approach generates inputs that (almost) fulfill the specification these programs use for data exchange, thereby implicitly passing these constraints.

Grammar-based Fuzzing. Like our approach, grammar-based fuzzers use a specification to generate valid inputs for programs, thus exercising deeper state space [21–29]. As opposed to our approach, typically these grammars must be manually generated. A subset of fuzzers attempts to approximate grammars without prior domain knowledge about the target [30, 34, 35]. While such approximations can identify logical units (chunks, tokens, or fields) within the targeted data format, they cannot solve complex constraints imposed on these logical units. This can also be seen in our evaluation, where FUZZTRUCTION outperforms WEIZZ for targets that use cryptography.

Domain Expertise. Given a human expert, they can bypass many of the challenges addressed by our work. A human expert can manually harness the target, remove checksums, provide a grammar to the fuzzer, or explicitly annotate the target to guide the fuzzer [20]. However, having a domain expert is costly and not feasible in all applications, such as legacy software. Our approach approximates the human expert's knowledge by harnessing the domain knowledge encoded by the programmer in the generator application.

Differential Fuzzing. Other approaches [27, 49–56] similarly exploit the fact that two programs share a specification: Differential fuzzers or, more general, differential testing. However, their underlying idea is to compare two consumers against each other, providing a fine-granular oracle that detects miscomputations beyond memory safety bugs. Our approach focuses on the two endpoints, a generator and a consumer, using a shared data format, making these approaches orthogonal. Our approach could be used to generate seeds that are then tested in a differential fuzzing setup of two consumers.

Mutation Testing. Similar to our approach, mutation testing [57–59] inserts *faults* into a target program. The goal is usually to simulate common programming bugs to assess the quality of a test suite or generate one. Our work is similar in that we inject faults into a program; however, we do not attempt to generate mutations that the test suites do not cover, but instead inject arbitrary faults. Instead of assessing the quality of an existing set of test cases, we use the outputs produced by the *mutant*, i.e., the buggy program, to fuzz another application. We then use coverage feedback within the fuzzing target, amongst other information, to determine the quality of a mutation. In summary, mutation testing and our approach share the idea of injecting faults; however, their goals and thus designs are fundamentally different.

8 Conclusion

In this paper, we present FUZZTRUCTION, a novel approach to software fault injection-based fuzzing. Based on the insight that programs that *consume* an input have one (or more) counterpart programs *generating* this input, we propose to instrument and mutate this generator. By injecting subliminal software faults, we can harness the implicit domain knowledge encoded in the application and generate inputs that *almost* match the specification. Using these inputs for fuzzing, FUZZTRUCTION produces high-quality inputs that produce high code coverage and exercise deep program states. Our approach is lightweight and does not require costly analyses or manually prepared execution environments. In the evaluation, we find that our approach shows its strength by generally outperforming state-of-the-art fuzzing methods, especially on targets with complex constraints, usually in the form of cryptographic primitives or compression applied to the input.

9 Acknowledgements

References

[1] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020. *(Cited on 1, 2, 7, 8, 13)*

[2] Ren Ding, Yonghae Kim, Fan Sang, Wen Xu, Gururaj Saileshwar, and Taesoo Kim. Hardware Support to Improve Fuzzing Performance and Precision. In *ACM Conference on Computer and Communications Security (CCS)*, 2021. *(Cited on 1)*

[3] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017. *(Cited on 1)*

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. RedQueen: Fuzzing with Input-to-State Correspondence. In *Symposium on*

Network and Distributed System Security (NDSS), 2019. *(Cited on 1, 3, 7)*

[5] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018. *(Cited on 1, 3, 5, 13)*

[6] Sebastian Poeplau and Aurélien Francillon. Symbolic Execution with SymCC: Don't Interpret, Compile! In *USENIX Security Symposium*, 2020. *(Cited on 1, 2, 3, 5, 7, 13)*

[7] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017. *(Cited on 1)*

[8] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *USENIX Security Symposium*, 2022. *(Cited on 1)*

[9] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *USENIX Security Symposium*, 2020. *(Cited on 1)*

[10] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christopher Kruegel, et al. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019. *(Cited on 1)*

[11] Karl Koscher, Tadayoshi Kohno, and David Molnar. Surrogates: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015. *(Cited on 1)*

[12] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021. *(Cited on 1)*

[13] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive Mutational Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2015. *(Cited on 1, 3)*

[14] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016. *(Cited on 1, 3, 5, 13)*

[15] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008. *(Cited on 1, 3)*

[16] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018. *(Cited on 1, 13)*

[17] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based Directed Whitebox Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2009. *(Cited on 1, 3, 13)*

[18] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy (S&P)*, 2010. *(Cited on 1, 3, 13)*

[19] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018. *(Cited on 1, 3, 13)*

[20] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. IJON: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020. *(Cited on 1, 14)*

[21] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *USENIX Security Symposium*, 2012. *(Cited on 1, 2, 14)*

[22] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2019. *(Cited on 1, 2, 14)*

[23] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019. *(Cited on 1, 2, 14)*

[24] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic Fuzzing with Zest. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019. *(Cited on 1, 2, 14)*

[25] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019. *(Cited on 1, 2, 14)*

[26] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy (S&P)*, 2020. *(Cited on 1, 2, 14)*

[27] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeong-won Kim, and Sukyoung Ryu. JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification. In *International Conference on Software Engineering (ICSE)*, 2021. *(Cited on 1, 2, 14)*

[28] Vasudev Vikram, Rohan Padhye, and Koushik Sen. Growing A Test Corpus with Bonsai Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2021. *(Cited on 1, 2, 14)*

[29] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *IEEE Symposium on Security and Privacy (S&P)*, 2021. *(Cited on 1, 2, 14)*

[30] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2020. *(Cited on 2, 3, 7, 13, 14)*

[31] Michał Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. *(Cited on 2, 18)*

[32] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017. *(Cited on 2)*

[33] OSS-Fuzz - Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. *(Cited on 2, 12)*

[34] Tim Blazytko, Cornelius Aschermann, Moritz Schloegel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019. *(Cited on 3, 14)*

[35] Wei You, Xuwei Liu, Shiqing Ma, David Mitchel Perry, Xiangyu Zhang, and Bin Liang. SLF: Fuzzing without Valid Seed Inputs. In *International Conference on Software Engineering (ICSE)*, 2019. *(Cited on 3, 14)*

[36] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *International Conference on Software Engineering (ICSE)*, 2022. *(Cited on 8)*

[37] lafintel. laf-intel - Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com. *(Cited on 8)*

[38] The DynamoRIO Team. DynamoRIO. https://dynamorio.org/. *(Cited on 8)*

[39] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018. *(Cited on 11)*

[40] Andrea Arcuri and Lionel Briand. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *International Conference on Software Engineering (ICSE)*, 2011. *(Cited on 11)*

[41] Andrea Arcuri and Lionel Briand. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014. *(Cited on 11)*

[42] Henry B Mann and Donald R Whitney. On a Test of whether One of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics*, 18(1):50–60, 1947. *(Cited on 11)*

[43] András Vargha and Harold D Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000. *(Cited on 11)*

[44] Robert J Grissom and John J Kim. *Effect Sizes for Research: A Broad Practical Approach*. Lawrence Erlbaum Associates Publishers, 2005. *(Cited on 11)*

[45] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *USENIX Security Symposium*, 2014. *(Cited on 13)*

[46] Dongdong She, Abhishek Shah, and Suman Jana. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2022. *(Cited on 13)*

[47] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: A Greybox Fuzzer for Network Protocols. In *International Conference on Software Testing, Validation and Verification (ICST)*, 2020. *(Cited on 13)*

[48] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *European Conference on Computer Systems (EuroSys)*, 2022. *(Cited on 13)*

[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011. *(Cited on 14)*

[50] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *IEEE Symposium on Security and Privacy (S&P)*, 2014. *(Cited on 14)*

[51] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014. *(Cited on 14)*

[52] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Symposium on Operating Systems Principles (SOSP)*, 2015. *(Cited on 14)*

[53] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed Differential Testing of JVM Implementations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016. *(Cited on 14)*

[54] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. Nezha: Efficient Domain-independent Differential Testing. In *IEEE Symposium on Security and Privacy (S&P)*, 2017. *(Cited on 14)*

[55] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *IEEE Symposium on Security and Privacy (S&P)*, 2021. *(Cited on 14)*

[56] Igor Lima, Jefferson Silva, Breno Miranda, Gustavo Pinto, and Marcelo d'Amorim. Exposing Bugs in JavaScript Engines through Test Transplantation and Differential Testing. *Software Quality Journal*, 29(1):129–158, 2021. *(Cited on 14)*

[57] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2010. *(Cited on 14)*

[58] Goran Petrović and Marko Ivanković. State of Mutation Testing at Google. In *International Conference on Software Engineering: Software Engineering in Practice*, 2018. *(Cited on 14)*

[59] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation Testing Advances: an Analysis and Survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019. *(Cited on 14)*

[60] Samples of binary with different formats and architectures. A test suite for your binary analysis tools. https://github.com/JonathanSalwan/binary-samples. *(Cited on 18)*

A Assigned CVEs

In Table 4, we list the CVEs assigned to bugs found by FUZZTRUCTION-NOAFL.

Table 4: Table of CVEs found by FUZZTRUCTION-NOAFL. We used FUZZTRUCTION-NOAFL rather than FUZZTRUCTION to ensure the found vulnerabilities cannot be attributed to AFL++'s mutations but are a result of our novel approach.

CVE identifier	Target	Bug Description
CVE-2021-4217	`unzip`	Out-of-bounds read in `fn`
CVE-2022-0530	`unzip`	Out-of-bounds read in `fn`
CVE-2022-0529	`unzip`	Out-of-bounds write in `fn`
CVE-2022-1304	`e2fsck`	Out-of-bounds write in `fn`

B Target Description

Table 5 describes the programs we have tested for our evaluation.

C Seed Description

Table 6 shows the different seeds sets we used as input for the generators and consumers during our evaluation.

Table 5: This table provides an overview of the different applications used during the evaluation. We tested the latest versions of the Ubuntu 20.04 packages available at the time of our evaluation. For `OpenSSL`, `vfychain`🔒, and `pngtopng`, we tested the latest versions from upstream.

Name	Version	Description
gendsa🔒	openssl 1.1.1l	An `OpenSSL` sub command used to generate (possibly encrypted) DSA keys
dsa🔒	openssl 1.1.1l	Sub command of `OpenSSL` for parsing (possibly encrypted) DSA keys
genrsa🔒	openssl 1.1.1l	A Sub command of `OpenSSL` to generate (possibly encrypted) RSA keys
rsa🔒	openssl 1.1.1l	Parsing of (possibly encrypted) RSA keys via the `rsa` sub command of `OpenSSL`
sign🔒	openssl 1.1.1l	The `req` sub command of `OpenSSL` for producing a self-signed keys certificate
vfychain🔒	vfychain 3.79	A utility that is part of the Network Security Services (NSS) suite of Mozilla Firefox and is used to validate certificate (chains).
7zip(🔒)	p7zip 16.02	Application used to compress or decompress data. Optionally encrypted and protected via password.
qpdf🔒 (🔒)	qpdf 9.1.1	Tool for encrypting and manipulating PDF files
pdftotext(🔒)	poppler-utils 0.86.1	Utility that is part of the `poppler` software suite and is used to convert (potentially encrypted) PDFs to text
zip(🔒)	zip 3.0	Decompressing utility with optional support for encryption
unzip(🔒)	unzip 6.0	Decompressing utility with optional support for encryption
pngtopng	libpng 1.6.37	Utility of `libpng` that simply parses a `png` into memory and writes it back to hard disk afterwards
e2fsck	e2fsprogs 1.45.5	A application for checking hard disk images for errors and inconsistency
objcopy	binutils 2.34	Utility for transforming different types of files, like ELFs. Used for, e. g., stripping symbols or removing sections.
readelf	binutils 2.34	Tool for dumping information of ELF files
objdump	binutils 2.34	Tool for disassembling ELF files and dumping additional information

Table 6: Table of the different seed sets used during fuzzing. The columns Generator and Consumer list the applications that used the described seed set in the role of a consumer or generator, respectively. In case of the generators, `objcopy` was used to generate inputs for two different consumers (`readelf`, `objdump`) which causes the number of generators (11) to be unequal to the number of consumers (12).

Seed set used for		Description	Set Name
Generator	Consumer		
{rsa🔒, dsa🔒, sign🔒}	{}	An empty seed set	empty
{mke2fs}	{}	A single file of size 256 KiB containing zeros	empty_256
{zip, 7zip, 7zip🔒}	{}	A single text file with the content aaaaa	text
{pdfseparate, qpdf🔒}	{pdftotext}	Six simple PDF documents containing forms, annotations, or basic geometric shapes	pdf
{}	{pdftotext🔒}	The same files as in the `pdf` set, but password protected	–
{}	{vfychain🔒 }	A self signed certificate generated using `OpenSSL`	–
{}	{rsa🔒}	A password protected RSA key pair generated by `openssl`	–
{}	{dsa🔒}	A password protected DSA key pair generated by `openssl`	–
{}	{7zip}	The `text` seed set compressed using `7zip`	–
{}	{7zip🔒}	The `text` seed set compressed and encrypted using `7zip`	–
{}	{unzip}	The `text` seed set compressed using `zip`	–
{pngtopng}	{pngtopng}	The `png` seed from AFL [31]	–
{}	{e2fsck}	The file of the `empty_256` set converted to an `ext4` image via `mke2fs`	–
{objcopy}	{objdump, readelf}	Files from [60] of size smaller than 1 MiB (a requirement enforced by AFL++) which `objcopy` is able to process	–