SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Structure-Aware Mutations for Library-Fuzzing

Valentin Metz

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Structure-Aware Mutations for Library-Fuzzing

Strukturwahrende Mutationen für das Fuzzing von Softwarebibliotheken

Author:	Valentin Metz
Supervisor:	Prof. Dr. Claudia Eckert
Advisors:	Fabian Kilger, Vincent Ahlrichs
Submission Date:	15.09.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.09.2023 Valentin Metz

Abstract

AutoDriver is an automatic generator for fuzz-drivers with the ability to pass complex objects between functions within a fuzz-target. However, its current implementation relies on the default, non-structure aware mutator suite of AFL++. This limitation results in a high occurrence of crashes caused by incorrect usage of the fuzz driver and a significant number of wasted fuzz cycles. We introduce a custom, structure-aware mutator for AFL++, specifically designed to generate the expected input structure of AutoDriver. By integrating this mutator, we expedite initial coverage exploration, enable deliberate utilization of AutoDriver-specific functionalities, and reduce the overall number of crashes logged due to structurally invalid input. To evaluate the effectiveness of our custom mutator, we compare it against the default mutator suite of AFL++ in a fuzzing-campaign with five well-known library targets. Our results demonstrate that the integration of a custom mutator presents a viable approach for enhancing AutoDriver's performance. We outline further research opportunities aimed at surpassing the performance of the default AFL++ mutator suite, enabling AutoDriver to produce better overall results in the library fuzzing domain.

Contents

1 Introduction

Fuzzing refers to the process of generating random input, which we then feed into the program under test, the *fuzz-target*. The fuzzer observes the target during execution and checks whether our input results in any unintended behavior or a crash. This is useful for automatic software testing, as we only need limited domain knowledge and do not need to write a complex suite of tests. Fuzzers are also a popular tool for security researchers, as a crashing program might indicate a security vulnerability [1]. By improving the capability of fuzzers to find bugs in complex software, we are more likely to find and fix potential security vulnerabilities before they get exploited.

Coverage-guided fuzzing enhances this approach by tracking execution path information inside the fuzz-target [2]. This enables us to track what effect our input has on the target program. Following this approach, we can gradually adjust our input by selectively mutating it, causing our executions to become more and more interesting. The part of the fuzzer that is responsible for generating these mutations is known as a *mutator*. A good mutator will apply a broad range of strategies to generate input, which is likely to have an interesting effect on the target [3].

Library fuzzing targets software libraries by calling user-exposed functions. As libraries are used in other programs, that in turn might expose these functions to user input, a vulnerable library function could leave an otherwise sound program at risk of exploitation due to a vulnerable dependency. Since libraries provide no executable that allows us to easily call all of these functions with arbitrary input, we manually write or generate [4] a fuzz-driver, that will take input and translate it into a sequence of function calls with input generated out of the input. Depending on quality and complexity of driver and library, not all input generated by the mutator constitutes a valid sequence of commands for the fuzz-driver.

A *structure-aware mutator for library fuzzing* describes a mutator, that is aware of the way input has to be generated and structured, so that the fuzz-driver can utilize it [5]. This can greatly increase the rate of input being generated having useful effects in the fuzz-target, as inputs that we know to be invalid are no longer generated. The mutator needs to consider what part of the input will be used to decide which functions will be called, and what part of the input will be used as input for those functions. Ideally, it should be able to change the input it generates in ways, that cause changes in some function calls but not others, so that the fuzzer can track and establish causality to

changes in coverage with changes in the input [3].

Current state-of-the-art fuzzers cannot easily generate complex function parameters in the context of library fuzzing, especially if one function depends on the output of another [5]. In this paper, we develop a structure-aware mutator that generates input for use with AutoDriver, a tool for the automatic generation of fuzz-drivers for library fuzzing, that can use the output of one library function as input parameter in another. Our mutator reduces the rate of crashes that are generated due to invalid input into the fuzz-driver. This allows for more effective crash evaluation by reducing the number of crashes due to library misuse.

2 Background

In this chapter, we explain the core concepts necessary for understanding our research. We explain coverage-guided fuzzing, the fuzzing process we rely on for exploring paths in our fuzz-target, and AFL++ which provides the toolbox we build our mutator upon. Further, we also explain the concept of a mutator, and why a fuzz-driver is needed for library fuzzing. We introduce the special case of AutoDriver, a system that automatically generates fuzz-drives with the ability to chain input. Finally, we give an overview of the serialization format used by our mutator in order to communicate with the fuzz-driver, and explain the type system it is based upon.

2.1 Fuzzing

Fuzz testing, also known as *fuzzing*, is a systematic and automated software testing technique employed to discover vulnerabilities and defects in computer programs, particularly those handling untrusted or malformed input data. In fuzz testing, a program is subjected to a barrage of deliberately generated, randomized, or mutated input data to assess its resilience and ability to withstand unexpected and potentially malicious inputs. By monitoring the program's behavior and identifying instances of crashes, unexpected errors, or security vulnerabilities, fuzzing helps software developers and security analysts pinpoint and rectify weaknesses, enhancing the overall robustness and security of the tested software systems. This technique has proven invaluable in identifying critical security flaws and improving software reliability, making it an essential tool in modern software development and security assessment [6].

2.1.1 Coverage-guided fuzzing

Coverage-guided fuzzing [2] attempts systematically uncover bugs in a program, without requiring domain-specific knowledge. To achieve this, a binary is instrumented with coverage information, allowing the fuzzer to track the path of execution a specific run of the program takes. Random input is then generated and incrementally mutated in a way that maximizes overall coverage of the program. Input that achieves higher coverage is prioritized as mutation input for future iterations. If a specific input causes the program to crash, that input is logged, allowing for detailed later analysis of the

bug. This leads to gradual exploration of a given program without requiring knowledge about implementation details [2]. As the process can be fully automated and is easy to parallelize, fuzzing can be used to automatically uncover bugs in large and complex programs. Large-scale fuzzing campaigns are undertaken by companies fuzzing their own products or open-source libraries, automatically filing issues for bugs found in the process [7].

2.2 AFL++

American Fuzzy Lop plus plus (AFL++) [2] is a state-of-the-art, stand-alone fuzzer and fuzzing framework. It provides compilers with support for instrumentation, an input selection and prioritization algorithm, and tools for crash triage. It supports a broad range of library-bindings, such as the Rust-LibAFL [8] bindings. As the fuzzing framework is composed of modular components, specific features and tools can be replaced, while maintaining interoperability with the other parts of the program necessary for a fuzzing-campaign. The mutators AFL++ uses by default can, for example, be replaced with a custom implementation that supports more complex mutations.

2.3 Mutators

The mutator is an essential part of mutation-based fuzzing [9]. It takes input from the fuzzer, applies a set of changes to it, and then returns the changed input back to the fuzzer, which then executes the fuzz-target with the generated input supplied as argument, file-input, or on stdin. This can either be done completely at random, by flipping bytes, adding values or by setting specific ranges to values that are deemed interesting, or output can be generated following a specific structure, making the mutator *structure-aware* [1]. Structure-aware mutators are especially useful if the fuzz-target is designed to work with a complex dataformat, such as parsers, compilers or image and video processing applications.

Different mutators can be combined as mutation-generators in a single fuzzing campaign, allowing the fuzzer to select well-performing mutators more often [3]. This does, however, require that all mutators adhere to the same structural requirements in regard to the input they accept and the output they generate.

Optionally, a trimming stage can be applied to keep the generated input short, while maintaining its interesting properties.

2.4 Fuzz-drivers

In order for a fuzzer to effectively test a target library, it must invoke the functions typically exposed to library users [10]. Fuzz drivers serve as intermediaries between the fuzzer and the target library, taking input data from the fuzzer (in our context, by reading a file provided by AFL++) and transforming it into a series of calls to library functions with corresponding input parameters. It is generally advantageous for the fuzz-driver to carry out this transformation in a consistent and predictable manner.

This predictability enables the fuzzer to optimize its path coverage by selecting well-performing inputs for further mutation. If it is not possible to call specific functions without disrupting the calls of other functions in the same fuzz-run, that can make the generation of complex input very difficult. Small improvements in one function call may unintentionally disrupt the structure of others. This highlights the importance of ensuring that the mutator component is aware of the input structure expected by the fuzz driver. By doing so, the mutator can generate and modify input data in accordance with this anticipated structure, thereby enhancing the overall fuzz-testing process [4].

2.4.1 AutoDriver

AutoDriver is a yet unpublished tool currently under development at the Fraunhofer Institute for Applied and Integrated Security (Fraunhofer AISEC).
AutoDriver automatically generates fuzz-drivers that offer control over the following three core concepts:

- Selection and order of functions to be called
- Initialization of function call arguments
- Management of target-library state

It has the unique ability to source function call arguments either from fuzzer-supplied input, or pass return values generated by one function as input parameter to another. This allows for the construction of arrays and pointers in the fuzzing process, which could otherwise not be generated by pure random mutation, as passing pointers and arrays require the memory area they point into to be allocated.

2.5 Serialization format definition

The serialization-format expected by the fuzz-driver depends on the fuzz-target. This must be communicated to the mutator, so that suitable input can be generated. We decided against recompiling our mutator anew for every fuzz-target, we supply it with a json file containing this information. Its location is passed to our library through an environmental variable and parsed on a call to `init()`. The exact information contained is as follows:

2.5.1 Set of types

All types used as argument or return type in one or more functions must be listed. A type is identified by a unique identifier `name` and mapped to a well-known internal type by `type`. Types can refer to other types, allowing the definition of arbitrarily complex objects. Structs, Typedefs and Unions can contain other types directly or alternatively contain Arrays or Pointers to other types.

```
{
  "type": "struct",
  "name": "struct s",
  "opaque": false,
  "fields": [
    "int",
    "int"
  ]
}
```

Listing 1: Example of a type definition

We also specify whether this type contains an opaque type, in which case we would not parse the type and instead mark it as `opaque`.

2.5.2 List of functions

All functions of the library that are intended to be called by a user of the library are listed here. They are represented as an ordered list of functions, where every function has a unique name, a return-type and a list of parameter types. Types are identified by unique name. If a type is only declared, but not defined (for example a library-internal struct that is transparently constructed inside a function and passed around by pointer), we refer to this type as opaque, in which case we will never pass this specific argument via the fuzz input section.

```
{
  "name": "simpleStructArgument",
  "return_type": {
    "type": "void"
  },
  "parameter_types": [
    {
      "type": "struct s",
      "opaque": false
    }
  ],
  "decision_bits": 2
}
```

Listing 2: Example of a function definition

Additionally, we supply the amount of decision bits we expect this function to have. This is not strictly necessary, as we are able to calculate the amount of decision bits assigned to a function from the types it is composed out of, but it provides useful debugging information, which we can compare against the amount of decisions bits expected to exist based on the type system.

2.5.3 List of chaining variables

An ordered list of types that are represented in the chaining input section of the fuzz-run control. As we do not actively use this feature, we have shortened this section to a single integer describing the minimal valid length this section can have if initialized to zero.

2.6 Fuzz-run control format

The fuzz-run control format is how we pass instructions on how to behave to the fuzz-driver. It consists of a variable length byte-vector passed to the fuzz-driver as input by AFL++. It is compartmentalized into 4 subsections:

1. Number of iterations
2. Decision bits
3. Chaining variables
4. Fuzz input

2.6.1 Number of iterations

The number of iterations is a single unsigned 16-bit integer, specifying how many fuzz-iterations we perform and by extent how many decision bits we should read. We currently call exactly one function per iteration.

2.6.2 Decision bits

The decision bits specify which functions should be activated in a given iteration, where their arguments should be sourced from and whether the returned input should be stored for future use by another function. They are represented as a fixed-size bitfield, serialized once per iteration, padded to the nearest byte-multiple.

Every callable function has at least one bit assigned to it, decing whether it should be called in this iteration or not. If the function has a return type that is chainable and not a function pointer, it gains an additional decision bit, specifying whether the returned value should be stored for future use. For every argument to the function that is chainable and not opaque, another decision bit is stored.

2.6.3 Chaining input

The chaining variables are a section of input shared between all functions that take or return the same datatype as input/output. The output of one function can be passed as input to another function, allowing the fuzzer to create complex datatypes and pointers, which could not have been initialized by pure random mutation alone. If functions return a pointer, that pointer can be stored and later used as input for another function. We have the ability to pre-initialize the chaining variable section, however, we decided against using this feature actively in our fuzzer, as we can simply supply our initial input directly through the fuzz-input section if desired. That is why we generally initialize the chaining input as null. As only pointers have corresponding chain-input, this initializes the length of the pointer-target-array to zero.

Chainable types

A type is considered chainable whenever it is not a BasicType or a typedef onto a BasicType. BasicTypes are all native C types that can not contain other types (int, float, void, unsigned long...). However, not all types that are considered chainable are actually chained due to current limitations in the fuzz-driver. Specifically, only pointers and arrays are valid types to chain, even though other types (like structs) still have a chaining bit. Additionally, we also consider function pointers a valid chainable type. These are not chained in the traditional sense of a set bit leading to input being taken from chaining input and an unset bit leading to input being read from the fuzz input. Instead, it leads to the argument being initialized either with a null-pointer, or a dummy-pointer managed by the fuzz-driver.

2.6.4 Fuzz input

The fuzz input contains the data passed as arguments to a function when called by the fuzz-driver. As we are fuzzing C programs, we need to be able to serialize and deserialize all valid C-data-types. For pointers, we specify a length, for unions, we specify the union variants. All types that contain other types are parsed recursively. Primitive types (void, int, char, unsigned long, float, etc.) are stored in their byte representation.

2.6.5 Function pointers

Function pointers have neither chain nor fuzz input. As they do have a decision bit, however, they are still included in the mutation process. If active, the fuzz-driver passes a dummy-pointer to the fuzz-target. Otherwise, a null-pointer is passed instead. This

is why they only have an assigned decision bit if passed as function-parameter and not if they appear as return value.

2.6.6 Opaque types

Opaque types are only declared, but not defined in the scope of the fuzz-driver. They are often used for library-internal types that operate on a black-box principle. Their exact contents are opaque, making it impossible for the fuzz-driver to construct them. They can therefore only be read from the chain-input, as the driver cannot instantiate them directly. That is why they do not have a decision bit assigned if passed as function-parameter, but do have an assigned decision-bit if returned from a function.

3 Concept

The original input format for AutoDriver was designed to work with the general-purpose default mutator of AFL++. Specifically, it had to be able to parse randomly generated, unstructured, malformed and incomplete input. It relied exclusively on the input selection mechanism of AFL++ to evolve the input in a way that would trigger specific features, such as calling a function with correctly formed parameters.

This causes a very large overhead in the fuzzing process, as most of the generated inputs have no meaningful impact on the fuzzing process as they are structurally invalid. Some mutations also change what function a specific section of input is assigned to, causing a causality loss between mutation and input that cannot easily be tracked by AFL. An example of this would be flipping a bit in the size specification for a pointer, increasing the number of bytes that will be interpreted and consumed as part of the pointer. This destroys the structure of all input bytes that follow, decreasing the efficiency of the fuzzing process.

We solve this problem by implementing our own, custom, mutator that is able to perform mutations, without having the mutations it performs interfere with each other in a disruptive way.

3.1 Mutator design

Starting from a single zero-byte seed, our mutator is invoked by AFL++ with a byte-vector input. We proceed to deserialize the input data to construct an internal model that encapsulates the ongoing active function calls. Subsequently, we execute multiple mutations on our internal representation and transmit the serialized outcome to AFL++ in the identical format as the initial input. AFL++ will then call the fuzz-target with our output, while monitoring coverage and identifying mutation output that produces noteworthy effects on the fuzz-target. The mutation output responsible for these effects is then conveyed back to our mutator, as input for further mutation iterations.

We generate and mutate input with respect to the structure AutoDriver expects, aiming to increase the coverage a fuzz-run can achieve with a given amount of CPU time. We give exactly as much input as needed for a given function call, if executed with the parameters we decided on. To achieve this,the mutator keeps track of every function call and parameter it emits with an internal representation, serializing function

calls and arguments recursively, and with the exact number of bytes that are required. Our mutations are well-defined and incremental, allowing for AFL to link a change in coverage to one mutation, which can then be refined (for example, by not performing unnecessary function calls), without destroying the effect deemed as interesting by AFL++.

We incorporate AutoDrivers ability to pass return values generated by one function as input parameter to another, deliberately enabling or disabling it for specific functions as part of the mutation process. This allows the fuzz-driver to call functions in the fuzz-target with input that could not have been easily generated by random mutation (such as pointers to sections of memory allocated on the heap).

The mutator adds, removes and mutates entire function calls at once. As we know the serialization format by which the driver parses fuzz-input, we never generate an invalid format, opposed to the default mutators of AFL++, which do, for example, not know about the exact number of bytes needed to fill the input byte vector read by the fuzz-driver.

For every function call, we can change the origin of where an argument is read from (chain or fuzz input), and in case it is read from the fuzz input (which we supply), we can modify the data it will contain. On aggregate types, we perform mutations recursively. Pointers and arrays can have their elements swapped, and in case their size is variable, elements added or removed. Alternatively, the mutation can be propagated downwards to mutate an element contained. For primitive data-types, we use the well-tested byte-field mutators of AFL++. We apply byte and bit flips, perform additions or subtractions, or set bytes to values likely to have an interesting effect.

4 Implementation

In this section, we detail how we interact with AFL++ and our fuzz-driver. We describe the serialization format definition used to establish inform our mutator about available data-types and the fuzz-run control format used to give instructions to the fuzz-driver in the fuzzing process. We also describe the actual mutations applied by our fuzzer.

As our mutator needs to support high throughput, as well as a complex type system, we decided to implement it in Rust. It is implemented as a library, which exports two functions callable by AFL++ [11]:

- init():
  Our initial setup function. It is called at the start of a new fuzz-run and establishes a list of functions callable by the fuzz-driver, as well as a list of all available types that can be passed as arguments to these functions. This is required so that the mutator can generate fuzz-input in accordance with format the fuzz-driver expects. The exact layout of the format used to achieve this, is described in Section 2.5.

- fuzz():
  This is the core mutation function. Whenever new input needs to be generated, it is called by AFL++. It receives a byte-slice of previous input as mutation base. We describe the serialization format, we use to relay our mutations to the fuzz-driver in Section 2.6. The actual mutations we apply, are discussed in Section 4.2.

4.1 Internal type representation

Internally, we represent a slight superset of all valid C-data-types:

Array: Fixed size vector of other type-objects.
All type-objects contained must be of the same type.

Pointer: Variable size vector of other type-objects.
All type-objects contained must be of the same type.

OpaquePointer: Represents a pointer to a type that cannot be instantiated by us.

Struct:	Fixed size vector of other type-objects. Type-objects contained can be of different types.
Enum:	32-bit unsigned int representing a C-style enum. We do not know which of the assignable integers actually map to a valid type.
Union:	Combination of a vector of type-objects and an index on that vector. The index specifies which of the possible union variants is currently selected.
Typedef:	Simple mapping to another type-object.
FunctionPointer:	Representation of a C-style function pointer. Cannot be instantiated by us.
BasicType:	All scalar types of C (int, double, unsigned short, char...). Represented as a fixed-size vector of bytes.

4.2 Mutation

We provide an overview of the mutations we perform, how they apply to each datatype, what effect we expect them to have and how they differ from the default mutators of AFL++.

4.2.1 Function calls

Each fuzz-run is composed of a list of function calls. In our mutator, it contains a reference to the function that gets called, information about whether to chain the return type (and whether it can be chained at all) and a list of arguments passed to the function:

```
struct FunctionCall<'a> {
    function: &'a Function,
    chain_return_type: Option<bool>,
    arguments: Vec<FunctionArgument>,
}
```

During one mutation, a function call can be added, removed or mutated. If we add a function call, we randomly select one functions from the list of all available functions and initialize it with default arguments (all arguments set to zero; pointers point to empty array). Similarly, if we decide to remove a function call, we simply delete one

function call from the list of function calls we emit towards the mutator. If we decide to mutate, we randomly select one of the function calls that we received as mutation input and change its properties. If the return type is chainable, we randomly set its bit. Then we randomly choose one of the function arguments from its `arguments` list and mutate it as described in the following section.

4.2.2 Function arguments

A function argument is a data type, that gets passed as parameter to a function in a function call. We differentiate between four flavours of `FunctionArguments`:

```
enum FunctionArgument {
    Basic(BasicType),
    FuzzInput(Type),
    Chained,
    PermanentlyChained,
}
```

A `Basic` function argument is every argument that contains a `BasicType` (see Section 4.1). `Basic` arguments cannot be chained and are therefore ignored during the `FunctionArgument` mutation stage. Similarly, a `PermanentlyChained` argument is used whenever an `OpaquePointer` is passed as argument. Since we can't initiate it, it is permanently chained, and we are not able to perform any mutations on it.

Every other data type is initialized as `FuzzInput`. If the type is an `Array`, `Pointer`, or `FunctionPointer`, it can be changed to a `Chained` argument in the mutation process.

Instead of performing a mutation on the `FunctionArgument` itself, we can also perform the mutation on its contained data type.

4.2.3 Data types

Every data type has a unique set of mutations available. Those data types that contain other data types (`Array`, `Pointer`, `Struct`, `Union` and `Typedef`) can also propagate mutations downwards in a recursive manner.

```
enum Type {
    Array(Array),
    Pointer(Pointer),
    OpaquePointer,
    Struct(Struct),
    Enum(Enum),
    Union(Union),
```

```
8       Typedef(TypeDef),
9       FunctionPointer,
10      BasicType(BasicType),
11  }
```

In an `Array` we can swap individual elements. In a `Pointer`, we can additionally increase or decrease the amount of elements contained. As `Structs` can not be changed themselves, we always propagate the mutation downwards to one of its elements. `Enums` are represented as a single 32-bit integer. We do however, not know which or how many of its assignable values constitute a valid enum variant, therefore set it to a random number whenever we call mutate on the enum. As `Unions` can contain multiple types, but only instantiate one of them at a time, we randomly select one of its representable types on its mutation. `TypeDefs` exclusively propagate mutations to its containing type.

For `BasicTypes`, we utilize a broad range of bit and byte level mutations sourced from the byte-field mutators of AFL++. Specifically, we:

- Flip a random bit
- Flip a random byte
- Increase a random byte
- Decrease a random byte
- Negate a random byte
- Add a random u8, u16, u32 or u64 to a randomly selected range of bytes
- Set a randomly selected u8, u16 or u32 to an interesting value
- Copy a random byte onto another random byte
- Set a randomly selected range of bytes to the same randomly selected value
- Copy a randomly selected range of bytes onto another randomly selected range of bytes
- Exchange two randomly selected byte ranges

5 Evaluation

In this chapter, we explain the experimental setup and approach used to compare our mutator against the default mutator suite of AFL++. We describe and evaluate the data of several fuzzing campaigns, and interpret its results.

5.1 Setup

In order to evaluate the efficiency of our mutator against the default mutator suite of AFL++, we have performed five fuzzing campaigns with each of the mutators, against five software libraries with known security issues.

Specifically, we used the following library-targets at the specified commit hash:

- libpng - `a37d4836519517bdce6cb9d956092321eca3e73b`:
  C-library for Portable Network Graphics (PNG) image files
- libsndfile - `86c9f9eb7022d186ad4d0689487e7d4f04ce2b29`:
  C-library for reading and writing files containing sampled audio data
- libtiff - `c145a6c14978f73bb484c955eb9f84203efcb12e`:
  C-library for reading and writing Tagged Image File Format (TIFF) files
- lz4 - `78433070abd9da36f225a5be1c302e42cdad67bd`:
  C-library providing a fast, lossless compression algorithm
- zlib - `21767c654d31d2dccdde4330529775c6c5fd5389`:
  C-library providing a lossless compression algorithm

All campaigns ran for 24 hours and were performed on an `AMD EPYC 7501 32-Core Processor` inside an `OpenStack-VM`. To ensure reproducible results and isolation, every campaign ran inside a Docker container. We used the `aflplusplus/aflplusplus` image as base, and limited every campaign to one processing core without hyper-threading.

5.2 Measured data

We consider the number of crashes saved by AFL++, as well as its reported overall map coverage.

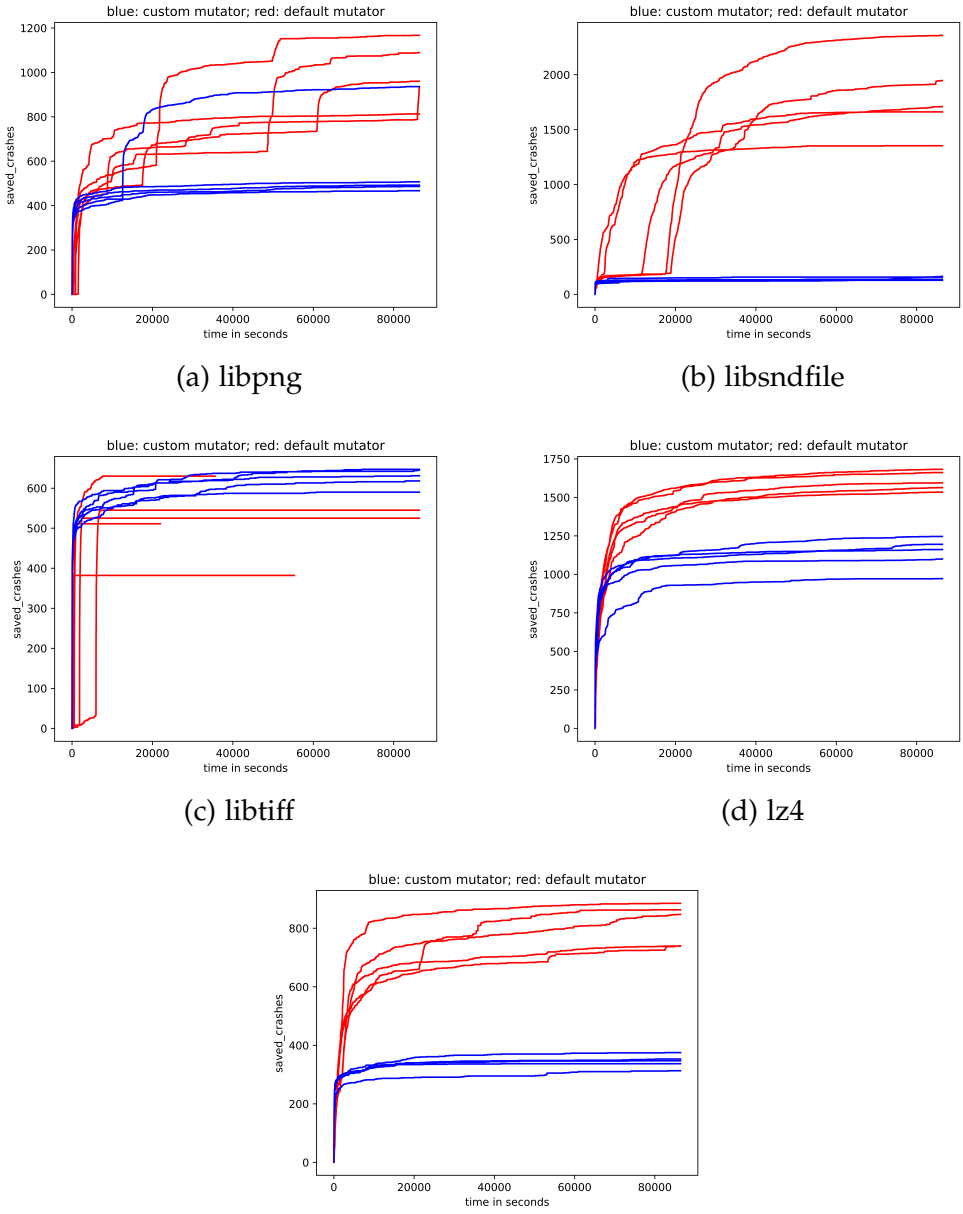
Crashes are only recorded by AFL++ if they provide a unique path of execution, with state transitions not yet seen in any previous crashes [12]. A state transition is composed of any two instrumentation points being hit in sequence [13]. As our target libraries contain only very few, known, bugs, we can infer that the majority of crashes are caused by supplying invalid input to library functions. So if a mutator were to produce fewer crashes, that would be an indicator for it producing less invalid input. Alternatively, it could also indicate a lack of coverage exploration, so we need to look at an additional metric to determine the cause of this behavior.

Map-coverage provides an indicator over the complexity of the target-library. It indicates how many of the possible execution paths are covered by the fuzzer during a fuzz-run, tracked by considering the number of instrumentation points that have been hit during execution of the fuzz-target. Low map-coverage indicating a more complex target-library, as well as potential roadblocks for the fuzzer [14]. A good mutator should be able to achieve greater map-coverage than a bad mutator, although for this measurement to be precise, we would need to exclude coverage in the fuzz-driver itself.

5.3 Results

Figure 5.2 shows edge coverage progression in percent over a 24-hour time period. Our custom mutator initially gains coverage faster the default mutators of AFL++. This behavior is, however, not sustained over time, and after approximately one hour, the default mutators surpass our custom mutator in terms of coverage. The plots for the number of saved crashes in Figure 5.1 show an almost identical pattern.

In the case of the `libtiff` library, we were able to generate more crashes than the default mutator suite, though this outcome is most likely due to a bug in the fuzz-driver, leading to an early crash in three of the five runs using the default mutator suite. Unfortunately, we were unable to address this issue in time for the evaluation.



(a) libpng

(b) libsndfile

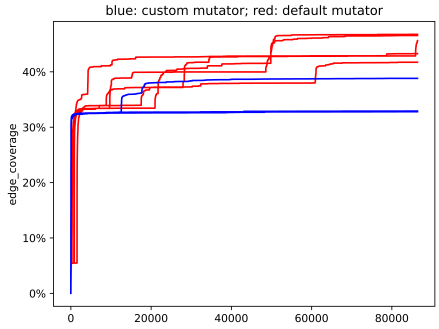
(c) libtiff

(d) lz4

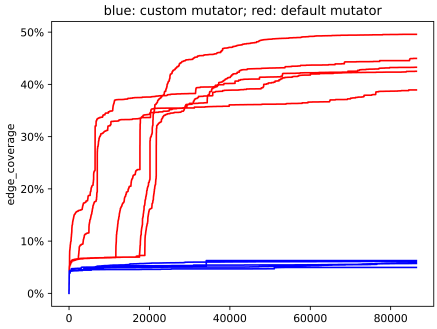
(e) zlib

Figure 5.1: Number of unique crashes recorded over time (24 hours) in multiple target libraries

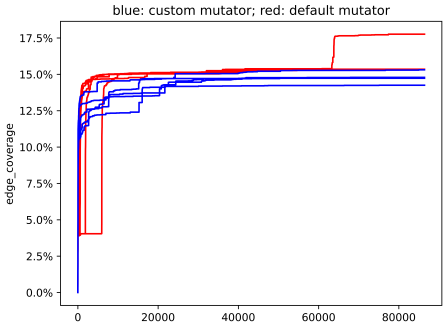
A crash is considered unique, if the associated execution paths involve any state transitions not seen in previously-recorded crashes [12]. A state transition is composed of any two instrumentation points being hit in sequence [13].



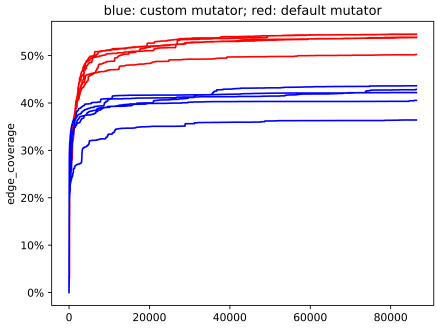
(a) libpng



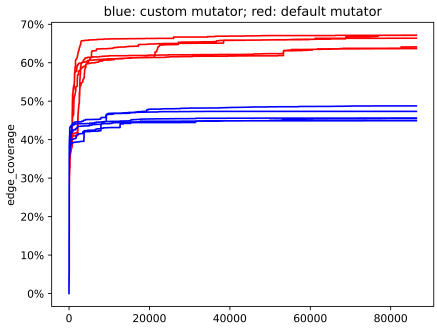
(b) libsndfile



(c) libtiff



(d) lz4



(e) zlib

Figure 5.2: Coverage of instrumented paths over time (24 hours) in multiple target libraries

5.4 Interpretation

Our ability to achieve initial coverage gains at a faster rate than the default mutators serves as a robust indicator of our mutator generating the fuzz-driver's expected input structure correctly. Even though this suggests that our mutator is operating in accordance with its intended purpose, it does *not* manage to outperform the default mutators in terms of coverage.

As the graphs for the number of saved crashes and edge coverage align so closely, this indicates that the lower number of crashes we generate is also not due to a more effective generation of valid input, but rather due to a lack of coverage exploration.

In fact, we cannot make any conclusive statements about the quality of our mutator's output, as this would either require a manual analysis of the crashes generated by both mutators, or more detailed coverage information than AFL++ currently provides, in order to separate coverage in the fuzz-driver from coverage in the target library. Both approaches are currently beyond the scope of this thesis.

We predict that the lower coverage-exploration of our mutator is due to its lack of support for the complete feature set offered by the default mutators of AFL++. Notably, features such as trimming and splicing are absent in our mutator, as well as the ability of AFL++ to select a specific mutation strategy more often than others [3].

Even though we were not able to surpass the performance of the default mutator suite in this thesis, we still recommend a continuation of this approach. Once we achieve feature-parity with the default mutator suite, we should offer a strict superset of its functionality, and we would expect our mutator to provide better coverage-exploration.

6 Future work

Our mutator does currently not support the full feature set of the default AFL++ mutator suit. Implementing the following features could improve the effectiveness of our mutator, allowing AFL++ to allocate resources more efficiently by focusing on more important/interesting input sequences.

Trimming Our mutator possesses the capability to eliminate function calls during the mutation process. Ideally, this task should be executed through a dedicated trimming stage. This enables AFL++ to selectively retain the most relevant portions of a specific mutation, thereby speeding up the fuzzing process, by executing the trimming stage before continuing the mutation of an interesting test-case [15].

Without a dedicated trimming stage, the fuzz-input can fill up with irrelevant data, slowing down the fuzzing process.

Splicing The process of splicing involves the amalgamation of features extracted from two distinct fuzz-runs, with the goal of potentially merging their noteworthy attributes. In our specific context, it is imperative that the splicing procedure maintains structural awareness, and merges runs on a function-call level, opposed to a mere byte-level fusion, as conventionally performed by AFL++ [15].

An intriguing avenue for research lies in investigating the methodology and sequencing of splicing our input. One plausible approach entails systematically executing function calls from one fuzz-run, followed by those from the other, while another strategy might involve interleaved execution. Alternatively, the content of aggregate types could be merged while preserving the original order of function calls. Each of these approaches warrants a comprehensive evaluation of their effectiveness.

Compartmentalizing the mutator Currently, we provide all our mutations in one mutator where they get activated by random chance. We could instead provide multiple mutators to AFL++ that are bound to specific mutations. AFL++ would then gain the ability to strategically select specific mutations that have been deemed especially effective, potentially leading to better usage of our fuzzing resources [3].

Recursive argument resolution Instead of randomly chaining arguments, we could attempt to resolve the precise order of function calls necessary to instantiate a specific argument. This would allow us to generate more complex input sequences, without the need to rely on the random chance of a specific function call being executed, which happens to return and store the desired argument on our chain-input [14].

Crash root cause analysis Presently, the predominant source of crashes generated by our fuzzing framework results from instances of library misuse. Addressing and mitigating those crashes, which stem from improper library utilization, is imperative for enhancing the efficacy of our mutator. By discerning and rectifying crashes originating from library misuse, we can proactively curtail the generation of analogous input sequences in subsequent fuzzing iterations. This necessitates modifications to the fuzz-driver component, which would need to inform our mutator of constraints that must be satisfied by input, in order to avoid triggering a particular invalid crash [14].

7 Related work

In this chapter, we outline several alternative approaches taken by other researchers in the field of structure-aware library fuzzing.

GraphFuzz GraphFuzz [5] targets C/C++ libraries with an automatically generated fuzz driver according to a handwritten specification. The specification needs to list all functions, classes and structs that are to be fuzzed. It utilizes a dataflow graph to track lifetimes and resolve object dependencies between functions, ensuring that functions with input dependencies are only called after those dependencies have been created. This allows for functionality comparable to the chaining-input of AutoDriver.

WEIZZ WEIZ [16] targets structured binary formats with a grey-box fuzzing approach.
Their fuzzing process is split into two stages:

1. Establish input to behavior causality and tag input sections with their possible effects (surgical stage)

2. Perform mutations on related/connected sections of input (structure-aware stage)

In the surgical stage, they attempt to establish input to program behavior causality, by grouping byte-fields of input into "chunks", whose change of content causes tracked and potentially deemed relevant instructions in the program to be executed or not. They achieve this by deterministically flipping bits in every input byte and logging the effect this has on execution flow. Every byte is then tagged with the changes it was able to cause.

The structure-aware stage utilizes this knowledge, by grouping chunks that are connected by tag during the mutation stage. Grouping and mutating input in sections is similar to our own approach, although we can establish precise bounds between sections of input, as our mutator is aware of the exact structure the fuzz-driver expects.

HOPPER HOPPER [14] is a zero-knowledge library fuzzing framework that performs grammar-aware mutations by utilizing an interpreter to establish constraints for function input. HOPPER attempts to generate valid function calls by using static analysis in

order to automatically select suitable predecessor function calls, that create the complex return type required as input for the target function in a recursive manner.

Input is mutated type-aware and under consideration of incrementally learned constraints, ranging from *non-null* for pointers to requiring specific array sizes and integer ranges that prevent overflows. The type-aware mutations they perform are closely related to the mutations we perform in our mutator. Another similarity is their concept of "nontrivial pointers", encompassing opaque-pointers, void-pointers, and function-pointers, which closely resembles our own type-system. Additionally, their mutator is able to communicate with their interpreter *during* a fuzz-run, in order to establish constraints that improve the quality of generated input. Applying this concept to our mutator would require changes in AutoDriver, but it could potentially be a viable approach to further improve its effectiveness.

8 Conclusion

In this paper, we introduced a custom mutator for AFL++ that works with AutoDriver to generate structure-aware mutations for library fuzzing. We described the design and implementation of our mutator, as well as its interaction with AutoDriver, that enables it to chain function calls, in order to pass complex objects between them. We validate our approach on five library targets, directly comparing it against the performance of the default AFL++ mutator suite. While we do not surpass the overall coverage exploration of the default mutator suite, we demonstrate that our custom mutator implementation is working as intended and can potentially be used to improve the performance of AutoDriver in the future. We outline further steps necessary to surpass the performance of the default AFL++ mutator suite.

Abbreviations

AFL++ American Fuzzy Lop plus plus

List of Figures

Bibliography

[1] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, *Smart greybox fuzzing*, 2018. arXiv: 1811.09447 [cs.CR].

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

[3] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966, ISBN: 978-1-939133-06-9.

[4] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "Intelligen: Automatic driver synthesis for fuzztesting," *CoRR*, vol. abs/2103.00862, 2021. arXiv: 2103.00862.

[5] H. Green and T. Avgerinos, "Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1070–1081. DOI: 10.1145/3510003.3510228.

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[7] G. LLC. "Oss-fuzz - continuous fuzzing for open source software." (), [Online]. Available: https://google.github.io/oss-fuzz/.

[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "Libafl." (), [Online]. Available: https://crates.io/crates/libafl.

[9] R. Gopinath, P. Görz, and A. Groce, "Mutation analysis: Answering the fuzzing challenge," *CoRR*, vol. abs/2201.11303, 2022. arXiv: 2201.11303.

[10] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: Fuzz driver generation at scale," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[11] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "Afl custom mutator documentation." (), [Online]. Available: https://aflplus.plus/docs/custom_mutators/.

[12] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "Afl documentation." (), [Online]. Available: https://afl-1.readthedocs.io/en/latest/fuzzing.html.

[13] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "Afl technical details." (), [Online]. Available: https://aflplus.plus/docs/technical_details/.

[14] P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen, *Hopper: Interpretative fuzzing for libraries*, 2023. arXiv: 2309.03496 [cs.CR].

[15] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "Afl fuzz approach." (), [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/afl-fuzz_approach.md.

[16] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: automatic grey-box fuzzing for structured binary formats," *CoRR*, vol. abs/1911.00621, 2019. arXiv: 1911.00621.