GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs

Harrison Green
hgarrereyn@forallsecure.com
ForAllSecure
U.S.A.

Thanassis Avgerinos
thanassis@forallsecure.com
ForAllSecure
U.S.A.

ABSTRACT

We present the design and implementation of GraphFuzz, a new structure-, coverage- and object lifetime-aware fuzzer capable of automatically testing low-level Library APIs. Unlike other fuzzers, GraphFuzz models sequences of executed functions as a dataflow graph, thus enabling it to perform *graph-based mutations* both at the *data* and at the *execution* trace level. GraphFuzz comes with an automated specification generator to minimize the developer integration effort.

We use GraphFuzz to analyze Skia—the rigorously tested Google Chrome graphics library—and benchmark GraphFuzz-generated fuzzing harnesses against hand-optimized, painstakingly written libFuzzer harnesses. We find that GraphFuzz generates test cases that achieve *2-3x more code coverage* on average with minimal development effort, and also uncovered previous unknown defects in the process. We demonstrate GraphFuzz's applicability on low-level APIs by analyzing four additional open-source libraries and finding dozens of previously unknown defects. All security relevant findings have already been reported and fixed by the developers.

Last, we open-source GraphFuzz under a permissive license and provide code to reproduce all results in this paper.

ACM Reference Format:
Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510228

1 INTRODUCTION

Fuzzing has become the de-facto standard for identifying new security vulnerabilities and ensuring software reliability. From common open source libraries to browser component testing [1] and from safety critical systems to automotive and aerospace standards [2], the entire industry is adopting fuzzing and finding thousands of critical issues before they occur in production.

The constantly increasing need for better automated testing has led to the development of a taxonomy of fuzzer types. *Coverage-guided* (grey-box) fuzzers such as libFuzzer [3] have gained significant adoption and visibility within the industry. These fuzzers consume feedback information from the target (edge coverage, value coverage, or similar) to guide test case selection and mutation. Coverage guided fuzzing is an *optimization problem*: the goal is to discover a corpus of test cases that maximizes coverage for a target. Intuitively, maximizing coverage leads to edge cases, erroneous behavior, and/or security vulnerabilities.

At the same time, the art of *model-based* (or structure-aware) fuzzing is growing rapidly as it offers two significant benefits. First, it enables fuzzing targets which expect complex, structured inputs. At the simplest level, a model-based fuzzer may simply unpack or post-process the input byte string. For example, the LLVM project [4] provides *FuzzedDataProvider.h* (FDP): a utility header file that splits a single fuzzer input into multiple smaller inputs. More complex model-based fuzzers, such as libprotobuf-mutator (LPM) [5], use custom generators and mutators to fuzz structured objects like *Protocol Buffers*.

Second, model-based fuzzers realize efficiency improvements over analogous unstructured fuzzers by avoiding bad inputs. Targets that validate their input before proceeding (i.e. through the use of checksums) are effectively un-fuzzable without manual countermeasures such as disabling validation at fuzz-time. Unstructured fuzzers simply get stuck trying to brute force input after input. Model-based fuzzers bypass this problem entirely by programmatically synthesizing valid test cases. For example, a checksum-aware fuzzer needs only to *compute* and *set* the correct checksum for a given input.

Despite the recent surge in fuzzing research, there is a noticeable lack of systems capable of fuzz-testing C/C++ libraries. Existing grey-box fuzzers such as libFuzzer [3] are particularly well suited for fuzzing one or two endpoints at a time but require manual effort (using FDP for example) to scale to multiple endpoints at once. CSmith [6] can synthesize realistic C code, but recompiling at each iteration is expensive when the target is a C *library* and not the C *compiler*. FUDGE [7] is a promising meta-fuzzing technique that automatically generates *harnesses* by analyzing and slicing a seed corpus of client-side code; however, it relies on Google's internal infrastructure and is not open source.

To address this gap, we introduce the concept of dataflow graph-based fuzzing in which a Library API interaction is represented as a dataflow graph. We describe algorithms for dataflow graph mutation, generation and execution in the context of C/C++ libraries. We open-source our implementation of dataflow graph-based fuzzing called GraphFuzz under a permissive license and we demonstrate its effectiveness by finding bugs in real-world targets and quantitatively benchmarking its performance against state-of-the-art

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510228

harnesses in the Skia Graphics Library. Additionally, we briefly survey the field of model-based API fuzzers to compare recent works and understand the benefits and drawbacks of various systems.

Overall, GraphFuzz makes the following contributions:

1) Model-based API Fuzzer Survey. We present a taxonomy for model-based fuzzers developed until today and show where GraphFuzz fits within the design space.

2) Dataflow graph-based fuzzing. We formally define dataflow graph-based fuzzing and introduce algorithms for performing graph mutation and generation in the context of coverage-aware fuzzing.

3) GraphFuzz for C/C++. We introduce our open-source implementation of dataflow graph-based fuzzing called GraphFuzz that is capable of semi-automatically fuzz-testing C and C++ libraries. We validate this technique by finding real world bugs and quantitatively benchmarking its performance against current state-of-the-art harnesses.

Section 2 provides a taxonomy for model-based API fuzzers and describes how GraphFuzz fits into the design space. We formally define the concept of dataflow graph-based fuzzing in Section 3 and in Section 4, we describe our open-source implementation called GraphFuzz. Section 5 evaluates GraphFuzz on real-world targets. We discuss limitations in Section 6 and the paper concludes in Section 7.

2 MODEL-BASED FUZZING

Model-based fuzzers (sometimes called structure-aware or grammar-based) use a *model* to inform test-case generation and mutation. The model constrains generated test cases and the search space of the fuzzer. When used correctly, models enable fuzzers to generate *interesting* inputs more efficiently than an unstructured fuzzer.

2.1 Model Domain

Model-based fuzzers specify (often implicitly) a *model domain* ($\mathcal{D}$) that describes the space of possible fuzzer test cases. This domain describes the internal *structure* of each test case and therefore restricts the types of mutations possible and the ways in which test cases can be invoked in a target.

As a baseline example, unstructured fuzzers use a *bytes* domain, i.e., each test case is a *byte sequence* and fuzzers can apply *byte-level* mutations such as swapping bytes, inserting substrings or mutating bytes. Libprotobuf-mutator (LPM) [5], an example of a different model-based fuzzer, represents test cases internally as a *tree* (a Protocol Buffer object) and therefore uses a *tree* domain. Using custom mutators, LPM performs *tree-level* mutations such as adding or removing leaves and rearranging subtrees.

We identify four broad classes of model domains as follows:

(1) **bytes**: Domains that consist of raw byte sequences. (i.e. all unstructured fuzzers use this domain)
(2) **tree**: Domains that are tree-like. These inputs typically represent an abstract syntax tree and are generated with a grammar such as a context-free grammar.
(3) **sequence**: Domains that consist of a list of items. Each item in the list can contain additional metadata.
(4) **graph**: Domains that consist of vertices and edges. We describe the first implementation of graph-based domains in this work.

2.2 Anatomy of a Model-based Fuzzer

Model-based fuzzers consist of the following four functions which operate on their respective model domains:

(1) **Generation**: seed $\rightarrow \mathcal{D}$
(2) **Mutation**: $\mathcal{D} \times$ seed $\rightarrow \mathcal{D}$
(3) **Crossover**: $\mathcal{D} \times \mathcal{D} \times$ seed $\rightarrow \mathcal{D}$
(4) **Invocation**: $\mathcal{D} \rightarrow$ feedback

The *seed* enables pseudo-random, yet deterministic behavior. The *feedback* metric always consists of at least a binary *bug* signal (did the target crash or not?) and typically contains more fine-grained data such as specific AddressSanitizer [8] issues or code coverage.

Generation-based fuzzers use only the *Generation* function to synthesize new test cases. These fuzzers are most applicable in black-box environments where no coverage information is available or when it is too complicated to define a *Mutation* operation. For example jsfunfuzz [9] and CSmith [6] use hand-crafted rules to generate realistic JavaScript and C code respectively.

In grey-box environments, *mutation* is necessary to reap the benefits of coverage feedback. *Mutation-based* fuzzers define the *Mutation* and/or *Crossover* operators to synthesize new test cases by mutating and mixing existing cases from a corpus. Mutation-based fuzzers such as libFuzzer [3] have demonstrated substantial efficiency improvements over purely generation-based fuzzers. Coverage feedback has also been employed with success in model-based fuzzers such as Nautilus [10], a script language fuzzer, and Pythia [11], a REST API fuzzer.

2.3 API Fuzzing Methods

Most real-world targets contain more than one endpoint:

- a web application provides HTTP endpoints
- a kernel provides system calls
- a C++ library provides public functions

Fuzzing *individual* endpoints at a time is not sufficient to discover every bug in a target. Some erroneous behavior only arises from the interaction of *multiple* endpoints. *API fuzzers* attempt to solve this problem of fuzzing many endpoints at once.

While modern techniques are varied, we identify four broad methods of API fuzzing and use them to categorize API fuzzers at a high level:

Method 1: Harness. A standard grey-box harness can be configured to act as an API fuzzer through manual effort. For example, a developer can fuzz-test a C++ library by procedurally invoking functions inside a for-loop and/or switch statement. Typically, a raw byte sequence from an unstructured fuzzer is used to initialize these pseudo-random values. For example, in the *FuzzedDataProvider* (FDP) approach, the fuzzer byte sequence is interpreted as a *byte stream* and values are pulled from this stream to initialize variables inside the harness. Similarly, one can use *libProtobuf-mutator* (LPM) [5] in conjunction with a coverage-guided fuzzer such as libFuzzer [3] to build a tree-based API fuzzer. For example, in Chromium's AppCache fuzzer [12], the Protocol Buffer instance represents a sequence of IPC calls.

Method 2: Code-gen. Some API fuzzers synthesize and execute program source code. This approach is most feasible for script-based

languages such as JavaScript and Ruby which do not require an expensive compilation step before execution, however this approach has also been used to fuzz-test C compilers [6].

While these fuzzers can generate realistic syntax patterns through the use of context-free grammars or similar models, they often fail to produce high-level, semantically meaningful code. For example, Han et. al [13] noted that 99% of the test cases from jsfunfuzz [9], a popular JavaScript fuzzer, raise a runtime error after only 3 statements.

Method 3: Harness-gen. Rather than building harnesses by hand, it is also possible to create systems that generate harnesses automatically or with little manual effort. For example, IMF [14] traces syscall logs to identify dependencies and synthesizes C harnesses that can fuzz-test these syscalls. Similarly, FUDGE [7] and FuzzGen [15] analyze a large code-base of client-side C/C++ code and extract slices of code to create fuzzer harnesses. While these systems can generate varied harnesses, the API invocation structure within a single harness is *static* at fuzz-time and only the values change.

Method 4: Dynamic. In the *dynamic* approach to API fuzzing, each test case represents a full API interaction sequence. The fuzzer engine dynamically processes each test case, invoking endpoints one-by-one. For example, in RESTler and Pythia, each test case is essentially a *list* of HTTP requests. The key distinction between *dynamic* and *harness-gen* approaches is that in a *dynamic* fuzzer, the structure of API interactions is specified at fuzz-time (as part of the test case) which allows the fuzzer to control both the values *and structure* of API calls. While *code-gen* fuzzers can also change the structure of API calls at fuzz-time through *recompilation* (e.g. CSmith [6]), *dynamic* fuzzers such as Syzkaller [16] and GraphFuzz (this work) bypass this expensive recompilation step.

2.4 Recent Model-based API Fuzzers

Manes et al [17] have extensively surveyed the current field of fuzzing. In this work, we focus specifically on model-based API fuzzers and narrow down our criteria and categorization to provide a detailed comparison. In Table 1, we survey a collection of recent model-based API fuzzers and compare various features (explained below). The table is primarily organized by the *method* of API fuzzing (as described in Section 2.3).

2.4.1 Fuzzer Features. For each fuzzer we list the primary target type, the model domain $\mathcal{D}$ (as described in Section 2.1) and the type of model used to generate and mutate inputs.

2.4.2 Mutation Engine. In the first column group, we compare the method of generating and exploring inputs. For each fuzzer, we list whether it supports *generation*, *mutation*, *crossover* and collecting coverage information. Generation-based fuzzers such as jsfunfuzz [9] and CSmith [6] implement only the *generation* function while mutation-based fuzzers also implement the *mutation* and/or *crossover* functions. In addition, some fuzzers such as Nautilus [10] and GraphFuzz (this work) support grey-box, coverage-guided fuzzing, using feedback from the target program to guide test case selection and mutation.

2.4.3 API Conformity. For the purposes of API fuzzing it is useful to compare syntactic and semantic features to understand how well the fuzzer conforms to a target API specification. In the second column group we compare the fuzzers based on three constraint attributes:

- **Syntax**: The fuzzer produces inputs that conform to language syntax rules.
- **Endpoint Dependencies**: The fuzzer produces inputs that ensure endpoints with dependencies are invoked after their dependents.
- **Object Lifetimes**: The fuzzer manages object lifetimes and invokes explicit constructors and destructors.

Fuzzers that conform to syntax rules alone will primarily target the parsing or compilation stage of the target. These fuzzers may be *capable* of generating semantically meaningful inputs but it is not guaranteed, and a large percentage of fuzz-time will be spent on inputs that are rejected. This level of fuzzing is most-applicable for script-based languages that target interpreters such as jsfunfuzz [9], LangFuzz [18], IFuzzer [19] and Nautilus [10].

Fuzzers that understand endpoint dependency requirements can synthesize inputs where all endpoint arguments are satisfied. For example, CodeAlchemist [13] maintains a set of JavaScript *code bricks* with explicit inputs and outputs and stitches together inputs that obey dependency rules. Similarly, RESTler [20] understands REST API dependencies and invokes endpoints that *consume* specific parameters only after a request has been made that *generates* that parameter (for example, a `POST /foo` before a `GET /foo`).

Fuzzers that are lifetime-aware explicitly manage the lifetime of objects. This ability enables fuzzers to operate in environments without automatic memory management and enforce semantic lifetime constrains such as the use of managed pointers. Additionally, these fuzzers can identify issues such as memory leaks and use-after-free bugs without false positives.

2.4.4 Applicability. In the third column group, we identify whether each fuzzer requires an input corpus of seed data (i.e. language grammar examples, client-side code, API traces, etc) and whether each fuzzer is open-source.

3 DATAFLOW GRAPH-BASED FUZZING

We propose a new technique of dataflow graph-based fuzzing in which a *Library API* interaction is represented as a dataflow graph. We develop algorithms to generate and mutate dataflow graphs according to a *schema* and describe how to execute such graphs in the context of C/C++ libraries. The concept of dataflow graph-based fuzzing is not restricted to C/C++, and we believe that future works will apply this technique to new environments.

To introduce the concept of a dataflow graph, we first provide an example of a bug found by GraphFuzz in the Skia Graphics Library. Figure 1 contains a snippet of C++ code that triggers a heap-use-after-free in Skia. In this example, the `shrinkTofit` method frees memory that is being used by the `SkContourMeasureIter` object. In Figure 2 we show the *same bug* represented as a dataflow graph; in short, functions are vertices and objects are edges. The key concept in GraphFuzz is that these two representations are *equivalent*. We can invoke this test case by either compiling and running the C++ code in Figure 1 *or* by dynamically executing the dataflow graph in Figure 2.

Method	Fuzzer	Target	Domain	Model	Generation	Mutation	Crossover	Coverage	Syntax	Endpoint Deps	Object Lifetimes	Works w/o Seeds	Open-source
Harness	libFuzzer (FDP)	Library API	bytes	Procedural	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓
Harness	libFuzzer (LPM)	Library API	tree	Protobuf	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓
Code-gen	jsfunfuzz (2007) [9]	JavaScript	bytes	Procedural	✓	✗	✗	✗	✓	✗	✗	✓	✓
Code-gen	CSmith (2011) [6]	C Compilers	bytes	CFG	✓	✗	✗	✗	✓	✓	✓	✓	✓
Code-gen	LangFuzz (2012) [18]	Script Languages	tree	CFG	✓	✓	✗	✗	✓	✗	✗	✗	✗
Code-gen	Dharma (2015) [21]	Data	tree	CFG	✓	✗	✗	✗	✓	✗	✗	✗	✓
Code-gen	IFuzzer (2016) [19]	Script Languages	tree	CFG	✓	✓	✓	✗	✓	✗	✗	✗	✓
Code-gen	Nautilus (2019) [10]	Script Languages	tree	CFG	✓	✓	✗	✓	✓	✗	✗	✓	✓
Code-gen	CodeAlchemist (2019) [13]	JavaScript	sequence	API Spec	✓	✗	✗	✗	✓	✓	✗	✗	✓
Harness-gen	IMF (2017) [14]	Kernels	-	-	✓	✓	✗	✗	✓	✓	✗	✗	✓
Harness-gen	FUDGE (2019) [7]	Library API	-	-	✓	✗	✗	✗	✓	✓	✓	✗	✗
Harness-gen	FuzzGen (2020) [15]	Libary API	-	-	✓	✗	✗	✗	✓	✓	✓	✗	✓
Harness-gen	RULF (2021) [22]	Rust API	-	-	✓	✗	✗	✗	✓	✓	✗	✓	✓
Dynamic	Syzkaller (2015) [16]	Syscalls	sequence	API Spec	✓	✓	✗	✗	✓	✓	✗	✓	✓
Dynamic	RESTler (2019) [20]	REST API	sequence	API Spec	✓	✓	✗	✗	✓	✓	✗	✓	✓
Dynamic	Pythia (2020) [11]	REST API	sequence	RG	✓	✓	✗	✓	✓	✓	✗	✗	✗
Dynamic	GraphFuzz (2021)	Library API	graph	API Spec	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: A survey of recent model-based API fuzzers organized by method. [1] requires manual implementation

```
1 SkPath *path = new SkPath();
2 path->moveTo(0, 0);
3 SkContourMeasureIter *iter = new
      SkContourMeasureIter();
4 iter->reset(*path, false);
5 path->shrinkToFit();
6 delete path;
7 iter->next();
8 delete iter;
```

Figure 1: The textual representation of crbug.com/1134261: a heap-use-after-free in the Skia Graphics Library found by GraphFuzz.

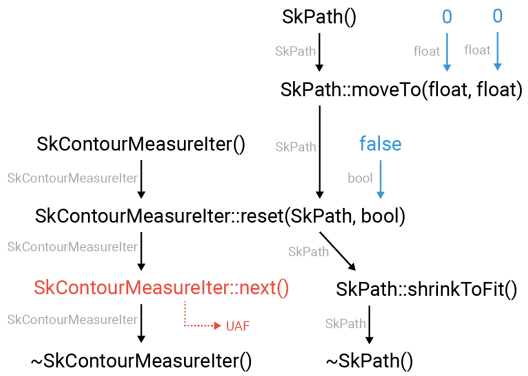

Figure 2: The dataflow graph representation of crbug.com/1134261 (Figure 1).

In this section we formally define the concept and terminology of a dataflow graph and we introduce our graph mutation and completion algorithms. In the following section we introduce GraphFuzz: an open-source framework for fuzzing C/C++ libraries with dataflow graphs.

3.1 Library API Specification

Library APIs provide two specifications for developers: Object declarations and Endpoint specifications that consume and produce the declared Objects.

Definition 3.1 (Object)**.** An object is the abstract specification of a datatype. We use O_x to denote an object domain and O to signify the domain of all valid objects.

Definition 3.2 (Endpoint)**.** An endpoint takes in a list of input objects and returns a list of output objects. For example, an endpoint $\mathcal{E}$ with x inputs and y outputs has a type signature of:

$$\mathcal{E} : O_1 \times O_2 \times \cdots \times O_x \rightarrow O'_1 \times O'_2 \times \cdots \times O'_y$$

We use the notation $\mathcal{E}^{(i)}$ to refer to the i'th input object and $\mathcal{E}^{(i')}$ to refer to the i'th output object.

Definition 3.3 (Endpoint Driver)**.** Each endpoint is associated with an endpoint driver that specifies exactly how to convert object inputs into object outputs. For C/C++ targets, the endpoint driver is a small function that is compiled into the harness.

Definition 3.4 (Library API)**.** A Library API defines a list of x Objects and y Endpoints: $API : O_1 \times O_2 \times \cdots \times O_x \times \mathcal{E}_1 \times \mathcal{E}_2 \times \cdots \times \mathcal{E}_y$

For a C/C++ API, objects consist of structs, classes, enums and primitive types while endpoints are generally methods. However, endpoints can be arbitrarily complex snippets of C/C++ code as described in Section 4.5.

3.2 Dataflow Graph

Definition 3.5 (Dataflow Graph)**.** The dataflow graph is a strongly-typed directed acyclic graph (DAG) that represents a specific, deterministic interaction pattern between endpoints.

$$\mathcal{G} : (V, E)$$

From a fuzzing perspective, each dataflow graph is equivalent to a traditional fuzzer test case. Vertices in the graph represent *instances* of endpoints and edges represent object dependencies: objects that are produced by one endpoint and consumed by another. For example, the edge: $E_i : (V_a, V_b, j, k)$ indicates that output j of V_a becomes input k of V_b. Dataflow graphs are *strongly typed*, so this edge is only valid if the object type is consistent, i.e. $\mathcal{E}_a^{(j')} = \mathcal{E}_b^{(k)}$.

A dataflow graph is *valid* if and only if every vertex has a single incoming edge for each object input and a single outbound edge for each object output. Additionally, there can be no directed cycles in the graph.

3.3 Invoking a Dataflow Graph

Each dataflow graph represents a deterministic, fully-formed interaction between endpoints. To invoke a dataflow graph, we generate an ordering of the vertices such that endpoints with object dependencies are invoked *after* the endpoints that produce those objects. Then, we iterate and invoke each endpoint driver in order, passing objects from one endpoint to the next as necessary.

For example, to execute the dataflow graph in Figure 2, we first invoke the constructor `SkPath()`, producing a new `SkPath` object. This object is passed to the next endpoint (`SkPath::moveTo`) which performs a method call on the object. We can't yet invoke `SkContourMeasure::reset` because we are missing a dependency, so we first invoke `SkContourMeasureIter()` to produce a new `SkContourMeasureIter` object. Only then can we invoke `SkContourMeasure::reset`, passing both the newly created `SkContourMeasureIter` object and the `SkPath` object. Execution continues in this manner until every vertex has been visited. Each vertex in the graph maintains a fuzzable *index* attribute to break ties in cases where the ordering is ambiguous.

3.4 Endpoint Context

For some targets, it is unwieldy to track all of the primitive values as discrete nodes in the graph. An endpoint that consumes an array of integers of size 100 would naively require 100 inbound connections, bloating the dataflow graph.

We simplify dataflow graphs by embedding certain objects directly into the graph vertex metadata. For C/C++ targets, we consider primitive types such as integers, floats and enums to be *short-lived*. These objects are not tracked as edges in the graph but rather their values are embedded directly into a graph vertex.

Specifically, we coalesce all fixed-size, short-lived objects for a given endpoint into a single *context byte string* that is stored as metadata inside a graph vertex. At fuzz-time, these primitive objects are initialized by deserializing the context byte string and provided to the corresponding endpoint driver.

Definition 3.6 (Context Byte String)**.** A context byte string consists of the concatenation of z fixed-size, short-lived types:

$$C = O_1^* \| O_2^* \| \ldots \| O_z^*$$

where $\|$ denotes concatenation of the raw byte representations of an object's value.

The *context size* is the number of bytes required to store all objects, which is simply the sum of each object's size:

$$|C| = \sum_{k=1}^{z} |O_k^*|$$

For example, $|O_k^*|$ is implemented as the `sizeof` operator for C/C++ targets.

Definition 3.7 (Optimized Endpoint)**.** An optimized endpoint consists of x long-lived inputs, y outputs and z short-lived inputs and has the following type signature:

$$\mathcal{E}^* : O_1 \times O_2 \times \cdots \times O_x \times C \rightarrow O_1' \times O_2' \times \cdots \times O_y'$$

Each vertex in the dataflow graph maintains an instance of an endpoint's context byte string in addition to the endpoint reference:

$$V : \mathcal{E} \times \mathcal{B}^{|C|}, \mathcal{B} \in \{0, 1, \ldots, 255\}$$

For example, in Figure 2, the short-lived types indicated in blue (two float inputs and one bool input) are initialized from a context string rather than a separate endpoint. The endpoint `SkPath::moveTo(float,float)` has an 8-byte context string which is deserialized into two 4-byte floats. The endpoint `SkContourMeasureIter::reset(SkPath,bool)` has a 1-byte context string which is deserialized to initialize the `bool` value (in this case, only the least-significant bit of this string is used).

3.5 Fuzzing Dataflow Graphs

The fuzzing process requires the ability to *generate* and *mutate* dataflow graphs. At the surface level, changing the structure of a dataflow graph (i.e. the vertices and edges) is straightforward. However, ensuring that each graph as a whole is still *valid* requires meeting specific constraints.

If a generated graph is invalid (missing edges or mismatched edge types), graph execution will produce false positive errors such as null pointer dereferences or memory leaks. For simplicity, we split the graph fuzzing problem into two parts, *graph mutation* and *graph completion*:

(1) **Mutation** (Section 3.6): Generate or mutate a dataflow graph to form an incomplete graph $\mathcal{G}'$ with potential missing edges.
(2) **Completion** (Section 3.7): Add vertices and edges as necessary to form a complete, valid graph $\mathcal{G}$.

3.6 Graph Mutations

In this section we list the graph mutations used by GraphFuzz. These specific mutations are not required for dataflow graph-based fuzzing, however in practice, we find them effective for the types of schemas used in C/C++ libraries. Each mutation acts on a graph G to produce an intermediate (potentially incomplete) graph G'.

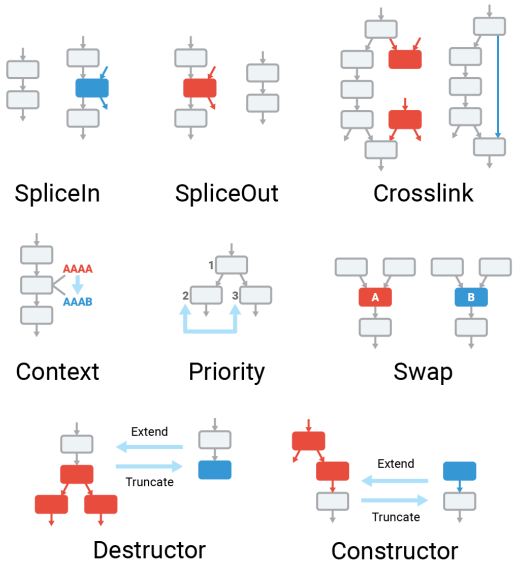

Figure 3: GraphFuzz Mutations

Following every mutation, the graph completion algorithm (Section 3.7) acts on G' to produce a valid, fully-formed graph G^*. A graphical representation of these mutations is shown in Figure 3.

3.6.1 Graph Mutations.

(1) **SpliceIn**: Splice a new endpoint between two existing endpoints. Given an existing edge (V_a, V_b, i, j), add a new vertex V_k and replace the old edge with two new edges: $(V_a, V_k, i, *)$ and $(V_k, V_b, *, j)$.
(2) **SpliceOut**: The opposite of the SpliceIn mutation. Given, a pair of edges $(V_a, V_k, i, *)$ and $(V_k, V_b, *, j)$ where $\mathcal{E}^{(i')} = \mathcal{E}^{(j)}$, remove the pair of edges and V_k and add a direct edge from V_a to V_b: (V_a, V_b, i, j).
(3) **Crosslink**: Sample two vertices V_a and V_b at random such that $\mathcal{E}^{(j')} = \mathcal{E}^{(k)}$ for some j, k. Remove the existing edges $(V_a, *, j, *)$ and $(*, V_b, *, k)$, and add a new internal edge between V_a and V_b: (V_a, V_b, j, k). If this mutation splits the graph into multiple disconnected parts, keep only the subgraph containing V_a and V_b.
(4) **Context**: Sample a vertex V_k with a non-zero sized context byte string ($|C_k| > 0$) and invoke libFuzzer's builtin mutator (`LLVMFuzzerMutate`) on C_k.
(5) **Priority**: Sample two vertices V_a and V_b at the same layer and swap their vertex indexes, reversing vertex priority during graph execution.
(6) **Swap**: Sample a random vertex V_a and replace it with a different endpoint of the same signature. By definition, the two endpoints are compatible with the existing graph structure so no further structural modification needs to be done.
(7) **TruncateDestructor**: Sample a vertex V_k such that $\mathcal{E}_k = (\cdots \rightarrow \cdots \times O'_j \times \dots)$. Remove the edge $(V_k, *, j, *)$ and if this splits the graph into two disconnected parts, keep only the subgraph with V_k.
(8) **ExtendDestructor**: Sample a vertex V_k such that $\mathcal{E}_k = (O \rightarrow \emptyset)$. Replace this vertex with a new vertex V'_k such that $\mathcal{E}_k = (\cdots \times O_j \times \cdots \rightarrow \dots)$ and replace the edge $(*, V_k, *, 0)$ with a new edge $(*, V'_k, *, j)$.
(9) **TruncateConstructor**: Sample a vertex V_k such that $\mathcal{E}_k = (\cdots \times O_j \times \cdots \rightarrow \dots)$. Remove the edge $(*, V_k, *, j)$ and if this splits the graph into two disconnected parts, keep only the subgraph with V_k.
(10) **ExtendConstructor**: Sample a vertex V_k such that $\mathcal{E}_k = (\emptyset \rightarrow O')$. Replace this vertex with a new vertex V'_k such that $\mathcal{E}_k = (\cdots \rightarrow \cdots \times O'_j \times \dots)$ and replace the edge $(V_k, *, 0, *)$ with a new edge $(V'_k, *, j, *)$.

3.6.2 Graph Crossover. Given two graphs $\mathcal{G}_a$ and $\mathcal{G}_b$, invoke the `Crosslink` mutation on two vertices $V_a \in \mathcal{G}_a$ and $V_b \in \mathcal{G}_b$.

3.6.3 Graph Generation. Initialize a new graph G with a single, random endpoint $\mathcal{E}$ and invoke the graph completion algorithm.

3.7 Graph Completion

We now introduce the *graph completion* algorithm that is used in graph generation and mutation to complete a partial graph $\mathcal{G}'$. We reduce the graph completion problem into several smaller problems of satisfying missing edges in a graph. To this end, we propose the sub-problem of edge completion:

Definition 3.8 (Edge Completion)**.** Given an object output of type O^*, the goal is to generate a subgraph $\mathcal{G}^*$ that is valid except for a single missing input edge of type O^*. Note that the problem is symmetrical to the case where given an object *input* of type O^* we want to generate a subgraph missing the corresponding *output*.

An incomplete graph $\mathcal{G}'$ can be completed by invoking the *edge completion* algorithm for every missing input and output and linking the generated subgraphs.

A naive, probabilistic approach to edge completion such as randomly sampling viable endpoints tends to generate extremely large graphs or fails to return a solution at all (instead the graph grows endlessly). It is possible to enforce certain *cutoff* rules, such as sampling endpoints with fewer connections with a higher probability but such heuristics are graph schema-dependent and we empirically found that they do not always work. Additionally, the edge completion algorithm runs several times per mutation, which may itself run 4 or 5 times per fuzzer iteration; a fast implementation is critical for fuzzer performance.

Our GraphFuzz implementation of edge completion pre-computes every possible subgraph for a given target object O^* by performing a breadth-first search of depth k over the graph schema. The subgraphs are stored in an probability-encoded tree structure (referred to as a `TypeTree`). At fuzz-time, the edge completion algorithm can sample subgraphs for any object type in constant time. These pre-computed trees are cached on disk which speeds up parallel fuzzing. For example, in libFuzzer `-fork` mode, a newly spawned thread can instantly retrieve the subgraph solutions.

In GraphFuzz, this pre-computation step also performs schema validation. Warnings are displayed if there are any endpoints in the

schema which are *unreachable* or *unsatisfiable* (i.e. cannot exist in a valid graph).

3.8 Graph Minimization

Crashing test cases often contain unnecessary cruft that can be removed to obtain a minimal reproducer. Although simplistic, we find that by randomly invoking mutations and retaining only those graphs which exhibit the same crash and are smaller, we obtain dataflow graphs close to the size of hand-minimized examples.

4 GRAPHFUZZ FOR C/C++

In this section, we describe GraphFuzz: an implementation of dataflow graph-based fuzzing designed to fuzz-test C and C++ libraries. We have released this framework along with documentation and example code under https://github.com/ForAllSecure/GraphFuzz.

4.1 Overview

GraphFuzz consists of two parts:

- **libgraphfuzz**: A core framework written in C++ which performs dataflow graph mutations and is linked into the fuzz harness.
- **gfuzz**: A Python command-line tool used to generate harnesses files and perform miscellaneous tasks such as graph minimization and automatic schema extraction.

4.2 GraphFuzz Schema

The core of a GraphFuzz harness is the *schema*. A schema is defined in a human-readable YAML file and contains a list of the API endpoints and object types available in a Library API. Using the schema, GraphFuzz automatically generates the *exec* and *write* fuzzer harnesses.

An example, partial schema for the Skia SkPath API is shown in Figure 4. This harness found a UAF in SkContourMeasureIter (https://crbug.com/1134261). GraphFuzz understands C and C++ function signatures and often times, this is the only information needed to define an endpoint.

4.3 Harnessing

Harnessing a target with GraphFuzz requires the following 5 steps, visualized in Figure 5:

(1) **Instrumentation**: Compile the target library with fuzzer coverage. With `clang` for example, this just requires adding the `-fsanitize=fuzzer` flag.
(2) **Schema Inference** (optional): Using `gfuzz`, run the *schema extractor* tool to automatically extract classes, structs, enums, typedefs and methods from library source code into a schema. The generated schema is a starting point for further modification.
(3) **Manual Revision**: Applying an understanding of the Library API requirements, manually fix up the schema by adding/removing classes, adding functions or redefining the input and output types of a function.
(4) **Harness Generation**: Run `gfuzz` on the schema to automatically generate two versions of the harness: *fuzzExec* executes the dataflow graphs while *fuzzWrite* converts the dataflow graphs to plain C/C++ source code which can be recompiled externally.
(5) **Compilation/Linking**: Link both harness variants to the target library to produce native libFuzzer executables.

```
typedef_SkScalar:
  type: typedef
  name: SkScalar
  value: float

struct_SkPath:
  ...
  methods:
  - SkPath()
  - void moveTo(SkScalar x, SkScalar y)
  ...
  - void shrinkToFit()
  - void close()

struct_SkContourMeasureIter:
  ...
  methods:
  - SkContourMeasureIter()
  - SkContourMeasureIter(const SkPath &path,
    bool forceClosed, SkScalar resScale)
  - sk_sp<SkContourMeasure> next()
  ...
```

Figure 4: A partial GraphFuzz harness for the Skia SkPath API. This harness includes function signatures for methods on the `SkPath` and `SkContourMeasureIter` structs in addition to `SkVector`, `SkPoint` and `SkContourMeasure` (not shown).

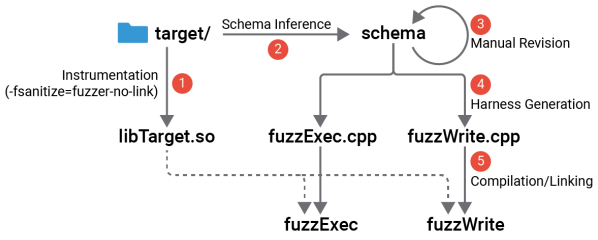

Figure 5: An overview of the GraphFuzz harnessing process.

4.4 Endpoint Driver Specification

In GraphFuzz, an endpoint specification is represented with the following four components:

- `inputs`: a list of endpoint input types
- `outputs`: a list of endpoint output types
- `args`: a list of context-based endpoint input types
- `exec`: an endpoint driver template (C/C++ code)

These attributes can either be defined manually or a developer can simply provide a C/C++ function signature and GraphFuzz will attempt to generate the endpoint specification automatically. In real world targets, we observe that full, manual endpoint definitions are

```
inputs: ["SkPath"]
outputs: ["SkPath"]
args: ["float", "float"]
exec: |
    $i0->moveTo($a0, $a1);
    $o0 = $i0;
```

Figure 6: A full endpoint driver specification for `SkPath::moveTo(SkScalar, SkScalar)`.

```
struct_SkPath:
  ...
  - int getPoints(SkPoint points[], int max):
      inputs: ["SkPath"]
      outputs: ["SkPath"]
      args: ["int"]
      exec: |
        SkPoint points[1024];
        unsigned int max = $a0 % 1025;
        $i0->getPoints(&points, max);
        $o0 = $i0;
```

Figure 7: A custom endpoint definition for `SkPath::getPoints(SkPoint[], int)`.

only required roughly 10% of the time. For the remaining cases, the function signature itself is sufficient.

The endpoint driver template (*exec*) is a snippet of C/C++ code that is exposed to several additional GraphFuzz-specific macros. Inside the endpoint driver template, `$iN`, `$oN`, and `$aN` refer to the N'th input, output, and argument respectively. Inputs and outputs have pointer types while arguments can be referenced directly as raw types.

For example, the `void moveTo(SkScalar, SkScalar)` signature defined on the `SkPath` struct in Figure 4 is converted into the full endpoint specification in Figure 6. GraphFuzz recognizes that this is a method call and automatically generates the correct input/output dependencies and execution driver template.

4.5 Custom Endpoint Drivers

Certain endpoints have implicit requirements about usage that are not inferable from the function signature alone. In these cases, a developer can extend the *exec* template to customize the specification of an endpoint.

For example, the method `SkPath::getPoints(SkPoint[] points, int max)` is used to retrieve points from the underlying path object. The `points` array must have space for `max` entries and will be filled in during execution. While GraphFuzz cannot infer these constraints from the function signature, they can be manually defined in the GraphFuzz schema.

We incorporate this semantic knowledge into the schema by defining a custom endpoint as in Figure 7. In this endpoint, we manually allocate a `SkPath` array on the stack with size 1024. Then we invoke `SkPath::getPoints` with this array and a `max` parameter which is bounded to the range 0-1024.

4.6 Fuzzing Process

GraphFuzz is implemented as a custom mutation engine on top of libFuzzer [3]. Hence, the resulting binaries are native libFuzzer executables and are compatible with existing fuzzing infrastructure such as OSS-Fuzz. The fuzzing process (Figure 8) consists of the following steps:

(1) **Selection**: LibFuzzer selects inputs to mutate.
(2) **Mutation**: GraphFuzz interprets each corpus input as a serialized dataflow graph (section 3.2) and applies graph-level mutations (section 3.6).
(3) **Execution**: GraphFuzz executes the dataflow graph (section 3.3) by performing a dynamic traversal of the vertices, executing corresponding endpoint drivers as necessary. The *fuzzExec* harness executes actual target code while the *fuzzWrite* harness converts nodes to corresponding source code output.
(4) **Coverage Feedback**: Feedback from the target (in the form of edge coverage, value coverage, etc.) is collected by libFuzzer and used to guide corpus growth and selection.

5 EVALUATION

We used GraphFuzz to fuzz-test 5 real-world C and C++ libraries. In this section we discuss challenges and results from our fuzzing campaigns. We also include several examples of the types of bugs found with GraphFuzz.

During our research, we sought to establish a quantitative, head-to-head benchmark. This endeavor proved difficult simply because there are not many existing harnesses that can test a large set of API endpoints at once — most existing harnesses only test one or two endpoints at a time, and therefore GraphFuzz could easily get more coverage by invoking more endpoints. However, we were able to find 10 harnesses in the Skia project which fuzzed between 6 to 350 endpoints at a time. We used these harnesses to establish a quantitative benchmark that we discuss in more detail in Section 5.2.2.

We found hard bugs (i.e. segfaults, use-after-free, buffer-overflows) in 4 of the 5 libraries we fuzzed and we found soft bugs (internal assertion errors) in every library. Since most of these bugs require API control to trigger, they are naturally less likely to manifest as security vulnerabilities. However we did find potential security vulnerabilities in both Skia and RDKit.

5.1 Implementation

All of the fuzzing experiments described in the following sections were performed on a 128-core AMD EPYC 7601 cluster with 512 GB of RAM. Fuzzer harnesses were compiled in an Ubuntu 18.04 or Ubuntu 20.04 Docker container using clang-10. For exploratory fuzzer runs, we used between 1 and 32 cores in libFuzzer fork mode. The Skia benchmark was performed with 4 cores per harness.

We provide all of the harnesses and supporting code to reproduce these experiments in the GraphFuzz repository.

5.2 Skia Graphics Library

Skia [23] is a a mature, C++ graphics library, maintained by Google and used in high-profile projects such as Chromium and Android.

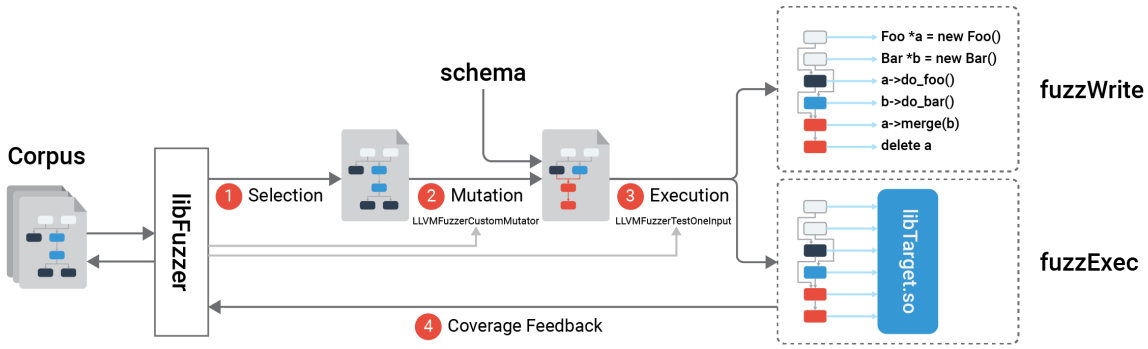

Figure 8: GraphFuzz Fuzzing process. 1. Corpus inputs are selected by libFuzzer 2. GraphFuzz applies semantic graph mutations 3. Inputs are invoked through a dynamic traversal of the dataflow graph 4. Coverage feedback is collected and used to guide corpus growth and selection.

The Skia codebase has been rigorously tested by Google security engineers, independent bug bounty hunters and the OSS-Fuzz project. Cumulatively billions of CPU hours have been spent fuzz-testing Skia.

As a graphics library, the core Skia API contains dozens of objects representing graphics primitives and thousands of functions that operate with these objects. Due to Skia's use in high-profile projects such as Chromium and Android, many of these internal APIs are potentially attacker-controlled. A malicious webpage can use specially crafted SVG or the JavaScript Canvas API to *induce* specific API calls in the renderer process.

During our research, we found hundreds of unique assertion errors. We also found and reported 3 security vulnerabilities in Skia. One we show in Figure 1.

5.2.1 Existing Harnesses. Due to its security importance, considerable time has been spent to develop fuzzing harnesses for Skia. Currently there are 37 separate libFuzzer targets in the Skia repository, most of which are fuzz-tested at scale as part of OSS-Fuzz. The majority of these harnesses isolate one specific endpoint such as a `deserialize` function or a `compile_shader` function.

Several of these harnesses use the *FuzzedDataProvider* approach to test a wide array of API endpoints at once. For example, the `fuzz_draw_functions` harness simulates drawing random shapes and paths to a canvas.

5.2.2 Head-to-head Benchmark. We selected 10 existing structure-aware, OSS harnesses from the Skia project as a baseline. For each existing harness, we created an equivalent GraphFuzz harness designed to fuzz *exactly* the same API surface. While in practice, it is not necessary to constrain GraphFuzz in this way, limiting the fuzzable API surface allows us to construct a fair head-to-head benchmark and ensure any differences in generated coverage are due to the *flexibility* and *efficiency* by which both variants can fuzz-test API interactions and not simply the *number* of fuzzable endpoints.

On average, for each GraphFuzz harness, we were able to accurately specify the usage semantics of 90% of target endpoints using *only* the function signatures. The remaining 10% of cases required minimal revision, for example to specify the usage of an array argument or constrain the domain of an input argument. See the "Auto" column in Table 2 for a breakdown by harness.

We performed 5 independent, 48-hour fuzz sessions for each harness using 4-cores in libFuzzer's fork mode. Each harness was linked against the exact same Skia build. After the designated period, we computed line coverage over the whole corpus using a gcov-instrumented version of each harness. For comparison purposes we compiled a second, instrumented version of each harness called the *dry harness* which is identical to the original harness except it does not invoke the target API. This *dry harness* allows us to account for differences in coverage due to the way both harnesses read and prepare test cases. For example, both the OSS and GraphFuzz harnesses use a small percentage of the Skia API to prepare data streams and set up the fuzz environment. We can use the difference in coverage between the normal harness and the *dry harness* to isolate only the code coverage that is due to actual target API fuzzing.

Specifically, given C_{oss} and C_{gf} (set of covered lines for OSS and GraphFuzz respectively) and the *dry harness* coverage: C'_{oss} and C'_{gf} (set of covered lines in the dry harness variants), we compute the shared fuzzer core coverage $R = C'_{oss} \cup C'_{gf}$ (i.e. the *uninteresting* code coverage due to harness-specific mechanics). Then we compute the normalized line coverage as $N_{oss} = |C_{oss} - R|$ and $N_{gf} = |C_{gf} - R|$.

To obtain temporal coverage information, we parsed the libFuzzer log output which contains both the elapsed time and a libFuzzer internal *cov* metric for many data points over the 48-hour period. This cov metric is not directly comparable between harnesses since it includes harness-specific coverage information. However, with the assumption that this metric scales roughly linearly with the true normalized line coverage (N_{oss} and N_{gf}), we graph a linearly-scaled version of the libFuzzer *cov* metric such that the final datapoint matches N_{oss} or N_{gf} exactly. This chart provides a visual cue for the evolution of the fuzzer corpora over time.

5.2.3 Benchmark Results. Fuzzer coverage is graphed in Figure 9 and we provide statistical information in Table 2.

In the best cases, the GraphFuzz harness generated nearly 9x as much line coverage compared to the baseline and in the worse cases, the GraphFuzz harness was roughly equivalent to the baseline.

In general, we see the largest coverage improvements with GraphFuzz on the larger harnesses. Intuitively, more endpoints means there are more opportunities for novel interactions. Manually defined harnesses miss these interactions unless they are specifically programmed to test them.

For small harnesses such as `api_regionop`, `api_pathop` and `api_path_measure`, the GraphFuzz harnesses are roughly equivalent to the existing Skia harnesses. Both harness variants quickly explore the majority of the state-space and plateau.

In larger harnesses such as `api_svg_canvas` and `api_draw_functions`, GraphFuzz continues to find new coverage until the end of the 48 hour period while the baseline harness plateaus early on.

5.3 Other OSS Targets

5.3.1 RDKit. RDKit [24] is a cheminformatics library written in C++ with Python bindings for most of the API. Typical usage involves constructing and manipulating many different objects such as `RDMol`, `RDAtom`, and `RDBond` (representing molecules, atoms, and bonds respectively).

RDKit has been continuously fuzzed as part of OSS-Fuzz since May 2020. Despite this, we found 10+ bugs with GraphFuzz (including heap-use-after-free and segmentation faults) after fuzzing a small portion of the RDKit API surface. Most of these bugs were also reachable from the Python API bindings. We disclosed three security relevant bugs to the RDKit developers.

5.3.2 SQLite. SQLite [25] is a small SQL database library written in C. It is used in Chrome, Android and hundreds of other projects. We spent a few days harnessing most of the SQLite3 C API with GraphFuzz. We identified two crashing test cases requiring 5 and 15 endpoints respectively. In the first case, GraphFuzz discovered that setting the `SQLITE_LIMIT_LENGTH` to 0 would crash a subsequent `sqlite3_prepare_v2` statement. In the second, GraphFuzz found a way to crash SQLite by invoking three online backups at once in a specific order. Due to the way SQLite is used in the wild, these bugs are unlikely to manifest as security vulnerabilities. However, for a heavily-fuzzed library like SQLite, it is impressive that GraphFuzz could discover these bugs.

5.3.3 Eigen. Eigen3 [26] is a C++ template library for linear algebra. It is used in projects such as TensorFlow and Chromium. We found this target particularly interesting to harness due to the extensive use of templates. Although GraphFuzz does not currently support C++ template syntax natively, it is possible to define schemas that use fixed-argument template functions. We used GraphFuzz to fuzz-test a subset of the matrix and vector API. Although we did not find any crashing bugs, we discovered test cases that reach dozens of unique assertions in the Eigen core library.

5.3.4 IOWOW. IOWOW [27] is a key/value storage library written in C. The provided API allows a user to create and destroy database objects and store/retrieve key-value pairs consisting of arbitrary byte-string data. We spent less than a day configuring a schema for IOWOW and fuzzing it and identified two crashing bugs within minutes of starting the fuzzer. In the first, GraphFuzz discovered that adding metadata to a database object with `iwkv_db_set_meta` and then destroying the database (`iwkv_db_destroy`) would cause a use-after-free upon closing the containing `IWKV` instance. In the second, GraphFuzz identified that initializing a database with the `IWDB_VNUM64_KEYS` flag and then invoking `iwkv_cursor_open` on the database using a lookup key of size 0 would trigger a segmentation fault. We reported both of these bugs to the maintainers and they were quickly patched in the latest version.

6 LIMITATIONS

6.1 Automation

While GraphFuzz can automatically synthesize a schema from a list of function signatures, we find that function signatures alone are not always sufficient to describe the usage requirements of an endpoint. For example, in the C function `void foo(bar *b)`, argument `b` could be an input, an output, or both. Similarly, in the function `void sum(int arr[], int N)`, there may be a "hidden" correlation between `arr` and `N` where the size of `arr` is expected to be at least `N`.

During our experiments, we observe that roughly 90% of endpoints (see table 2) can be accurately modeled with only the function signature while the remaining 10% require human-curated custom endpoint definitions. The development of systems which can automatically infer (or search) these implicit constraints in Library APIs is an interesting area for future research.

6.2 False Positives

Given an incorrect schema, GraphFuzz can generate false positive crashes do to invalid API usage. In other words, generated graphs are only as accurate as the provided schema. Users of GraphFuzz need to take care to ensure that the GraphFuzz schema aligns with the target Library API. We call this problem *schema alignment*.

In practice, we observe two distinct types of schema alignment issues:

6.2.1 Type 1: Single-endpoint Semantics. Given a schema with an incorrectly specified endpoint (such as the examples in Section 6.1), GraphFuzz will quickly and frequently generate false positive crashes. These issues are both easy to diagnose and easy to correct by manually redefining the endpoint.

6.2.2 Type 2: Multi-endpoint Semantics. Some Library APIs have hard-to-model usage requirements that span multiple endpoints. For example, one API pattern we observe in Skia is the use of a shared pointer `sk_sp<Foo>` along with a `Foo *refFoo()` method. The returned `Foo` pointer is valid as long as the original `sk_sp<Foo>` object has not been destroyed. In this case, modeling the usage requirements is more difficult than simply modeling each endpoint. A user needs to create a schema such that `sk_sp<Foo>` cannot be destroyed while a `Foo` pointer is still used elsewhere. For example, it is possible to define a *synthetic type* in the GraphFuzz schema which bundles the returned `Foo` pointer along with the original `sk_sp<Foo>` object such that the `sk_sp<Foo>` object cannot be destroyed until the `Foo` pointer is released.

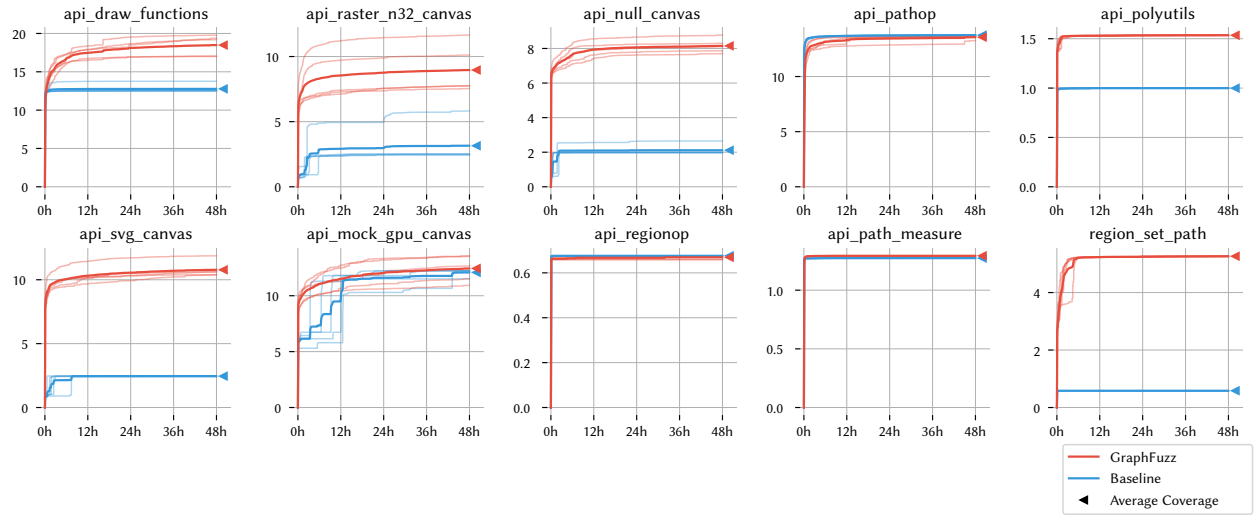

Figure 9: GraphFuzz vs. OSS-Fuzz harnesses on 10 Skia benchmarks. Each line shows a complete, 48-hour fuzz session on 4 cores. The x-axis shows elapsed time (in hours) and the y-axis shows normalized line coverage (in thousands). The bold lines (indicated by the marker) show the average coverage from GraphFuzz and OSS respectively across the 5 runs.

Harness	Endpoints	Auto	Baseline NLC	GraphFuzz NLC	Coverage Δ
api_draw_functions	72	90%	12,770 ± 497	18,486 ± 1,206	1.45x
api_raster_n32_canvas	350	84%	3,152 ± 1,336	8,964 ± 1,640	2.84x
api_null_canvas	350	84%	2,114 ± 267	8,152 ± 371	3.86x
api_pathop	19	100%	13,752 ± 7	13,568 ± 158	0.99x
api_polyutils	7	86%	999 ± 0	1,535 ± 1	1.54x
api_svg_canvas	350	84%	2,455 ± 23	10,778 ± 558	4.39x
api_mock_gpu_canvas	350	84%	12,089 ± 292	12,452 ± 1,061	1.03x
api_regionop	6	100%	676 ± 0	669 ± 8	0.99x
api_path_measure	17	100%	1,273 ± 0	1,293 ± 0	1.02x
region_set_path	53	85%	584 ± 0	5,264 ± 10	9.01x

Table 2: Skia benchmark results. NLC = Normalized Line Coverage, reported as (mean ± std) for 5 fuzzer runs. Coverage Δ = average increase in coverage gained by GraphFuzz over the baseline (1x means no change). Auto = percentage of endpoints specified using only the function signature.

This solution is effective at preventing false positives but is a stop-gap for a more complex problem and limits the flexibility of generated graphs. Further research is needed to design solutions that enable more accurate modeling of multi-endpoint semantics.

7 CONCLUSION

In this paper, we introduced the technique of dataflow graph-based fuzzing which is designed to fuzz-test Library API's. We describe this approach in the context of fuzzing C and C++ libraries and we release our implementation of dataflow graph-based fuzzing called GraphFuzz as an open-source framework. We validate our approach on five real-world targets and demonstrate that GraphFuzz can find real bugs and outperform hand-crafted harnesses in quantitative benchmarks at a fraction of the development cost. Often times, the only information required to fuzz with GraphFuzz is a list of function signatures in a target.

REFERENCES

[1] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," 2017.
[2] "Fuzzing for safety critical systems." https://forallsecure.com/safety-critical. Accessed: 2021-09-03.
[3] K. Serebryany, "libfuzzer–a library for coverage-guided fuzz testing," *LLVM project*, 2015.
[4] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.
[5] "google/libprotobuf-mutator," June 2021. original-date: 2017-01-11T22:57:02Z.
[6] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 283–294, 2011.
[7] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 975–985, 2019.
[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pp. 309–318, 2012.

[9] J. Ruderman, "Introducing jsfunfuzz," *URL http://www. squarefree. com/2007/08/02/introducing-jsfunfuzz*, vol. 20, pp. 25–29, 2007.
[10] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars.," in *NDSS*, 2019.
[11] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations," *arXiv preprint arXiv:2005.11498*, 2020.
[12] "chromium/src.git - Git at Google."
[13] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines.," in *NDSS*, 2019.
[14] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2345–2358, 2017.
[15] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 2271–2287, 2020.
[16] D. Vyukov, "Syzkaller," 2015.
[17] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
[18] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st {USENIX} Security Symposium ({USENIX} Security 12)*, pp. 445–458, 2012.
[19] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *European Symposium on Research in Computer Security*, pp. 581–601, Springer, 2016.
[20] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 748–758, IEEE, 2019.
[21] "MozillaSecurity/dharma," May 2021. original-date: 2015-03-25T17:56:23Z.
[22] J. Jiang, H. Xu, and Y. Zhou, "Rulf: Rust library fuzzing via api dependency graph traversal," *arXiv preprint arXiv:2104.12064*, 2021.
[23] "Skia: The 2d graphics library." https://skia.org/. Accessed: 2021-09-03.
[24] "Rdkit: Open-source cheminformatics." http://www.rdkit.org. Accessed: 2021-09-03.
[25] "Sqlite: In-memory database." https://www.sqlite.org/. Accessed: 2021-09-03.
[26] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3." http://eigen.tuxfamily.org, 2010.
[27] "Iowow: C11 key/value database engine." https://iowow.io/. Accessed: 2021-09-03.