HOPPER: Interpretative Fuzzing for Libraries

Peng Chen
Tencent Security Big Data Lab
spinpx@gmail.com

Yuxuan Xie
Tencent Security Big Data Lab
ryanyxie@tencent.com

Yunlong Lyu
Tencent Security Big Data Lab
loydlv@tencent.com

Yuxiao Wang
Tencent Security Big Data Lab
yuxiaowang@tencent.com

Hao Chen
University of California, Davis
chen@ucdavis.edu

ABSTRACT

Despite the fact that the state-of-the-art fuzzers can generate inputs efficiently, existing fuzz drivers still cannot adequately cover entries in libraries. Most of these fuzz drivers are crafted manually by developers, and their quality depends on the developers' understanding of the code. Existing works have attempted to automate the generation of fuzz drivers by learning API usage from code and execution traces. However, the generated fuzz drivers are limited to a few specific call sequences by the code being learned. To address these challenges, we present HOPPER, which can fuzz libraries without requiring any domain knowledge to craft fuzz drivers. It transforms the problem of library fuzzing into the problem of interpreter fuzzing. The interpreters linked against libraries under test can interpret the inputs that describe arbitrary API usage. To generate semantically correct inputs for the interpreter, HOPPER learns the intra- and inter-API constraints in the libraries and mutates the program with grammar awareness. We implemented HOPPER and evaluated its effectiveness on 11 real-world libraries against manually crafted fuzzers and other automatic solutions. Our results show that HOPPER greatly outperformed the other fuzzers in both code coverage and bug finding, having uncovered 25 previously unknown bugs that other fuzzers couldn't. Moreover, we have demonstrated that the proposed intra- and inter-API constraint learning methods can correctly learn constraints implied by the library and, therefore, significantly improve the fuzzing efficiency. The experiment results indicate that HOPPER is able to explore a vast range of API usages for library fuzzing out of the box.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

Fuzzing, Vulnerability Detection, Automated Test Generation, Interpreter, Code Synthesis

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3616610

ACM Reference Format:
Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. HOPPER: Interpretative Fuzzing for Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616610

1 INTRODUCTION

Fuzzing is one of the most popular techniques for finding software vulnerabilities. Fuzzers give the software a large number of random inputs and observe whether unexpected behaviors happen. Though the idea is simple, fuzzing has successfully been applied to testing various applications and found many bugs. In recent years, the techniques have been greatly improved due to the advent of grey-box fuzzing. Coverage-based grey-box fuzzers such as AFL [44] and LibFuzzer [26] mutate inputs to explore deeper program states without requiring knowledge about input format or program specifications. Constraint-based grey-box fuzzers [9, 10, 14, 43, 35, 8], as the state-of-the-art fuzzers, employ constraint-solving techniques to reach code branches that are guarded by complex constraints.

While grey-box fuzzing techniques greatly facilitate the generalization and automation of program fuzzing (i.e., fuzzing techniques that take ready-to-use binary programs as targets), replicating such success in library fuzzing (i.e., fuzzing techniques that take APIs as targets) is challenging. To use the state-of-the-art fuzzers for libraries, users have to manually craft a fuzz driver that consumes the type-agnostic inputs fed by fuzzers and transforms the byte stream into API arguments. However, writing high-quality fuzz drivers is difficult, as it is time-consuming and requires a deep understanding of the library. Consequently, most of the existing fuzz drivers cover only a small part of library APIs. The APIs, especially the rarely used ones, usually lack adequate testing. Library fuzzing is still struggling to be usable out-of-box and scalable due to the lack of an automated solution.

In library fuzzing, fuzz drivers need to use correct argument types when invoking APIs and satisfy both intra- and inter-API constraints. Otherwise, the fuzz drivers would crash unexpectedly. For instance, to test `ares_send(ares_channel channel, char *qbuf, int qlen, ares_callback callback, void *arg)` from *c-ares*, a fuzz driver should initialize the first argument `channel` with a call of `ares_init(ares_channel *channelptr)` to meet the inter-API constraint, and then set the third argument `qlen` to be the length of the second argument `qbuf` and set the forth argument `callback` to be a non-null function pointer with the type of `ares_callback` to comply with the intra-API constraints. Any violation of these three constraints may result in spurious crashes. However, in practice,

information about these constraints is either missing or scattered across library documents or comments, making it hard to collect them in a fully automatic way.

Recently, researchers have proposed learning-based [6, 22, 45, 24] and model-based [18, 36, 21] methods for automatically generating fuzz drivers. Learning-based methods, such as FuzzGen[22], attempt to learn the correct usage of APIs from existing consumer code. However, this method fails when consumer code is unavailable, such as for new or work-in-progress libraries. Model-based methods, such as GraphFuzz [18], ask users to provide specifications of the APIs under test, which requires domain-specific knowledge and significant human involvement. Furthermore, the quality of the fuzz drivers generated with these methods is largely affected by the external inputs (i.e., consumer code or user-provided expertise), which can be inaccurate or incomplete. For example, in our experiments, we found that some of the fuzz drivers provided by the authors of FuzzGen [22] and GraphFuzz [18] result in spurious crashes due to misuses in the consumer code and incorrect user-defined schemas (Section A.2 in Appendix). The fuzz drivers [1] generated by FuzzGen only cover 5 of 26 API functions in the *libvpx* decoding library and only support the *vp9* codec. Similarly, the specifications [2] of *sqlite3* written by the authors of GraphFuzz do not even include commonly used API functions, such as `sqlite3_exec` and `sqlite3_complete`.

To address the aforementioned challenges, we present Hopper to fuzz APIs without requiring external knowledge. Inspired by coverage-based fuzzing, which learns valid format from mutating random seeds [29], Hopper learns potential usage of the APIs from mutating the composition of API calls and their arguments. If executing the mutated program triggers a new path or a new crash, Hopper infers the intra- and inter-API constraints based on the dynamic feedback. To achieve this, we introduce a Domain-Specific Language (DSL) that describes arbitrary API usages. The DSL inputs can be interpreted with a lightweight interpreter linked against the library under test. In this way, we transform library fuzzing into interpreter fuzzing. The fuzzer is now responsible for generating programs encoded in the format of DSL to feed into the interpreter. Then the interpreter executes the programs to see if unexpected behavior happens. Thanks to the grammar-aware mutation and the inferred constraints, Hopper is able to generate valid inputs that explore different API usage while excluding false positive crashes.

We implemented Hopper and evaluated its effectiveness on 11 real-world libraries against manually crafted fuzzers (MCF) and other automatic solutions (i.e., FuzzGen and GraphFuzz). Table 2 shows that Hopper greatly outperformed the other fuzzers in both code coverage and bug finding. Notably, Hopper improves the code coverage over MCFs in *cJSON* by 47.12% in line coverage and 37.50% in branch coverage and achieves higher coverage than the total of 9 fuzzers in *zlib*. In total, Hopper found 25 new bugs in the libraries, and 17 of them have been confirmed, as shown in Table 4. Moreover, we have demonstrated that the proposed intra- and inter-API constraint learning methods can accurately learn constraints implied by the library, thereby significantly improving the fuzzing efficiency.

2 BACKGROUND

2.1 Library Fuzzing

The impact of security vulnerabilities in libraries is significant due to their widespread use in various programs. However, testing libraries through program fuzzing alone is often inadequate. Library APIs may be invoked by programs under complex path constraints or with specific arguments that make it difficult to exercise the APIs fully. To address this issue, library fuzzing tools have been developed to specifically test libraries, with LibFuzzer being a common choice for such testing. Here are the required steps for effectively using LibFuzzer to fuzz libraries:

- **Craft a fuzz drivers**. A fuzz driver is a program that describes the usage of library APIs, including a sequence of API calls and their arguments. A high-quality fuzz driver should provide an entry to explore as much code in the library as possible for thorough testing. Specific execution paths are not only determined by the arguments in API calls but also by invoking their related APIs. These related APIs may return values as arguments or affect the context for other APIs that rely on them (e.g., global values). Hence, a deep understanding of the tested library is necessary for thorough testing. However, enumerating all valid API usages would be time-consuming and challenging. Therefore, fuzz drivers usually contain only a few common API usages. As shown in Table 1, the MCFs of the listed 11 popular libraries only covered 18.58% of the APIs, whereas the remaining 81.42% uncovered APIs will escape being fuzzed. Even though some approaches have been proposed to automatically synthesize fuzz drivers, the coverage of APIs is still limited (e.g., GraphFuzz [18] achieved API coverage of 41.42%). Moreover, since different API usages have different search spaces for fuzzing, putting all of them in sequence would be inefficient. Instead, it would be more effective to write them into multiple fuzz drivers or execute them conditionally within a single fuzz driver. For example, in Figure 2, the fuzz driver calls the parsing function and then calls different printing functions according to the first 4 bytes of `data`.
- **Specify the format of input**. The blind byte stream generated by LibFuzzer makes it difficult to create structured inputs that satisfy intra-API constraints. To address this issue, we need to specify the input format and guide the fuzzer to generate arguments beyond byte arrays. FuzzedDataProvider [21] is capable of dividing the fuzz input into multiple parts of various types, while libprotobuf-mutator[36] can generate structured inputs based on provided grammars. However, the presence of intra-API constraints makes defining the potential argument range for fuzzing an enormous task. Hence, developers might opt to encode arguments as literal constants directly into fuzz drivers, such as the second argument of `cJSON_ParseWithOpts` in Figure 2. Unfortunately, this approach can result in inadequate testing for API functions.

Table 1: Number of unique APIs used in fuzz drivers. The second column is the total number of exported APIs in libraries.

Library	Total	MCFs	FuzzGen*	GraphFuzz	Hopper
cJSON	78	6(7.69%)	4(4.13%)	40(51.28%)	78(100%)
c-ares	60	13(21.67%)	-	20(33.33%)	58(96.67%)
libpng	241	25(10.37%)	-	66(27.39%)	233(96.69%)
lcms	283	10(3.53%)	-	38(13.43%)	274(96.82%)
libmagic	18	4(22.22%)	-	10(55.56%)	18(100%)
libpcap	89	8(8.99%)	-	29(32.58%)	73(82.02%)
zlib	84	29(34.52%)	-	41(48.81%)	78(92.86%)
re2	70	35(50.00%)	-	47(67.14%)	69(98.57%)
sqlite3	279	15(5.38%)	-	74(26.52%)	225(80.65%)
libvpx	26	7(26.92%)	5(19.23%)	17(65.38%)	24(92.31%)
libaom	38	5(13.16%)	7(18.42%)	13(34.21%)	35(92.11%)
Average	-	**18.58%**	**13.93%**	**41.42%**	**93.52%**

*The authors of FuzzGen released the fuzz drivers for *libvpx* and *libaom*, and the released code of FuzzGen is unable to run on the rest libraries, except for *cJSON*.

2.2 Fuzzing Interpreters

Grammar-aware grey-box fuzzing has succeeded in programs that parse the inputs, especially in interpreters [34, 41, 3, 7, 19, 39, 25, 11]. The success of fuzzing interpreters is attributed to the following two key techniques.

- **Grammar-aware input mutation**. Blindly mutated inputs are likely to be rejected by the parsing procedure, while structured inputs can reach deeper paths. Grammar-aware mutations parse the inputs as intermediate representations (IRs) based on their encoding grammar and mutate the IRs while adhering to constraints. For example, Superion [41] uses an abstract syntax tree as the IR and conducts three main operators on the tree for mutation: replace a node in the tree with a newly created sub-tree, splice two different trees, and minimize trees without affecting execution.
- **Coverage guided fuzzing**. Guided by the coverage feedback, the fuzzer keeps inputs that trigger new paths and mutates them further to enter deeper branches or find new bugs. By incorporating grammar-aware mutation strategies, the fuzzer can effectively synthesize valid test inputs that cover more branches and trigger new bugs.

We observe that constructing fuzz drivers for libraries resembles implementing an interpreter for the inputs, and fuzzing the interpreter is equivalent to fuzzing the library under the hood. However, MCFs interpret type-agnostic byte inputs into typed values to feed limited sequences of API calls, which only partially explore the libraries. If we extend the interpreter to accept any API usages as input, a grammar-aware coverage-based fuzzer can efficiently generate inputs for the interpreter and thoroughly explore the libraries.

3 DESIGN

3.1 Overview

Hopper transforms the problem of library fuzzing into the problem of interpreter fuzzing. At a high level, Hopper consists of two main components: a grammar-aware fuzzer that generates inputs encoded in a DSL format, and a lightweight interpreter that executes these inputs, as shown in Figure 1.

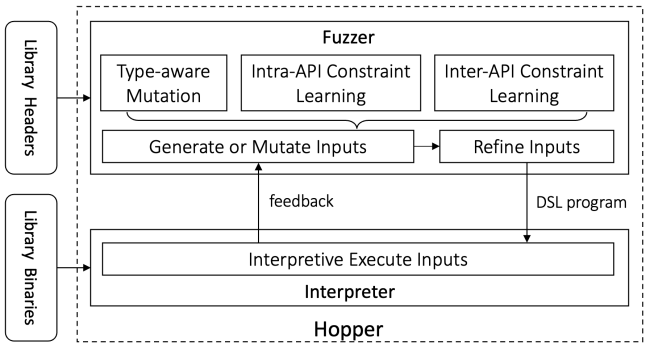

Figure 1: Overview of Hopper's architecture.

The fuzzer produces high-quality inputs for invoking library APIs. It first extracts function signatures and type definitions from the library's header files, which provide valuable information on potential relations between API functions and argument types. By leveraging this information, Hopper generates a sequence of API calls through random combinations of API functions and arguments. At the early stage, the generated sequences may be invalid and terminate with shallow paths in the APIs' code. However, thanks to coverage guidance, Hopper keeps the inputs that explore new branches as seeds and mutate them further. Ultimately, the generated sequences become valid and their execution can reach deeper code. Completely random fuzzing wastes enormous time in generating illegal and inefficient inputs. To accelerate the process of evolution, Hopper performs type-aware argument mutation (Section 3.3.2) and learns intra- and inter-API constraints (Section 3.4). Additionally, Hopper minimizes input size (Section 3.3.3) to reduce overhead.

The interpreter consumes the inputs and invokes the library APIs. The libraries under test are linked against the interpreter during the compiling stage. Before linking, Hopper instruments the binaries to capture internal states during execution, e.g., code coverage. When new input arrives, the interpreter parses the input according to the syntax of DSL and interprets the statements orderly based on their semantics.

3.2 DSL and Input Interpretation

Fuzz drivers are usually developed with the same programming language as the library's specification. Therefore, for compiled language like C/C++, fuzz drivers are crafted and compiled before fuzzing begins. Only the input bytes are changed during each round of fuzzing. To replace the sequence of API calls during fuzzing while avoiding compilation overhead, we introduce a Domain Specific Language (DSL) and a lightweight interpreter in Hopper for accelerating the entire process.

3.2.1 DSL. For the sake of generality, the interpreter in Hopper takes *DSL programs* as input. The grammar of Hopper DSL is listed in Figure 6 in the appendix. Each *DSL program* comprises *statements* as its most fundamental components, and each statement has a unique index that its successors can reference. We categorize

common fuzzing behaviors found in MCFs into five *statement* types in our DSL.

- A **load statement** defines the type information and literal representation of a value. Strong typing enables the generation of input data directly in a specific type, which eliminates the requirement of converting it from a type-agnostic byte stream.
- A **call statement** invokes a specific API function in libraries by providing the function name and a list of arguments, where each argument refers to the value defined in a previous load statement.
- An **update statement** overwrites the value returned by a call statement at runtime.
- An **assert statement** checks a call's return value at runtime. If the assertion fails (e.g., the caller needs to dereference the return value but the value is null), the program exits immediately.
- A **file statement** specifies a valid file resource for I/O operations. If the statement is used for reading, a sequence of random bytes is filled in the file for runtime reading.

To keep the grammar simple, our DSL does not support conditional statements. Some fuzz drivers use conditional statements to choose different API functions or arguments. By contrast, Hopper generates different DSL programs to enumerate those API functions or arguments. For example, Figure 2 is a real-world fuzz driver written by the developers of *cJSON*, and Figure 3 is a program written in DSL that covers a path in Figure 2.

3.2.2 Interpreter. The interpreter parses DSL programs, executes the statements gracefully, and monitors the states of the program after executing each statement. In order to invoke the library APIs, Hopper links the interpreter against the libraries under test at compiling stage. It also constructs a table that correlates each function's name with its corresponding caller based on the header files of the libraries. During program execution, the caller casts values to the types of arguments it needs and then invokes the referenced function with the arguments. Before linking to the interpreter, Hopper instruments the library binaries with code that counts branches, and hooks compare instructions and resource management functions (e.g., `malloc`, `free` and `fopen`). The interpreter then collects the following feedback at runtime.

- **Optional Branch Tracking.** It is unnecessary to track branches of all the library API calls in the DSL program every time, so Hopper defines a global flag to guard the branch tracking code. The value of the flag is determined by the DSL input. When calling an API function that ends with a question mark (e.g., Line 13 in Figure 3), the interpreter enables branch tracking for that call.
- **Context-sensitive Code Coverage.** To distinguish the same branches visited by different APIs, the interpreter sets the hash of the current API function's name as the context. The instrumentation code reads the context and calculates an exclusive code trace for each API.
- **Overflow Detection.** If the statement loads a variable-sized value (e.g., an array), the interpreter of Hopper stores them in a memory arena and appends a canary right after it to detect possible buffer overflow.
- **Use-after-free Detection.** The interpreter maintains a set of memory chunks allocated by `malloc` and released by `free` through instrumentation. For each pointer used in function arguments, if the memory chunk pointed by this pointer is freed, the interpreter exits the program immediately to avoid the use-after-free issues.
- **Comparison Hooking.** The interpreter collects the parameters used in comparison instructions and functions to guide the fuzzer to solve magic bytes.

```
1  int LLVMFuzzerTestOneInput(const uint8_t* data, size_t
       size) {
2    cJSON *json;
3    size_t offset = 4;
4    unsigned char *copied;
5    char *printed_json = NULL;
6    int minify = data[0] == '1' ? 1 : 0;
7    int require_termination = data[1] == '1' ? 1 : 0;
8    int formatted = data[2 ]== '1' ? 1 : 0;
9    int buffered = data[3] == '1' ? 1 : 0;
10   if (size <= offset) return 0;
11   json = cJSON_ParseWithOpts((const char*)data + offset,
         NULL, require_termination);
12   if (json == NULL) return 0;
13   if(buffered) {
14     printed_json=cJSON_PrintBuffered(json, 1, formatted);
15   } else {
16     if(formatted) printed_json = cJSON_Print(json);
17     else printed_json = cJSON_PrintUnformatted(json);
18   }
19   if(printed_json != NULL) free(printed_json);
20   if(minify) { ... }
21   cJSON_Delete(json);
22   return 0;
23 }
```

Figure 2: Fuzz driver written by the developer of cJSON. The code is in cjson_read_fuzzer.c, which is used by OSS-Fuzz [31].

The fuzz drivers utilized by LibFuzzer should have no side effects since the code runs in a loop in the same process. However, it is difficult to generate a program that can reset all resources before exiting. Hopper's interpreter solves this problem by running each input in an individual process. Once a DSL program terminates, the operating system destroys the process and releases all allocated resources. This allows the interpreter to execute DSL programs continuously without the need to release resources.

3.3 Grammar-aware Input Fuzzing

To find bugs in libraries rather than interpreters, Hopper takes a different approach compared to other grammar-aware fuzzers. While other fuzzers traverse all possible combinations of syntaxes based on input grammar, Hopper instead focuses on generating various effective API calls. This process involves two phases, as shown in Figure 4. First, Hopper generates inputs based on the information available in function signatures to initialize a seed

```
<0> load Vec<char>= vec(32)["
     GXsAAAAAAAAAo9tsrXXoqw57jwAAAAAAAAARNk+1AAA="]
<1> load char* = &<0>
<2> load char** = null
<3> load int = 0
<4> call cJSON_ParseWithOpts (<1>, <2>, <3>)
<5> assert non_null(<4>)
<6> load cJSON = { next: null, prev: null, child: null,
     type_: 8, valuestring: null, valueint: 12345,
     valuedouble: 0.2771, string: null }
<7> update <4>[0.child] = <6>
<8> load Vec<char> = vec(7)[54, 52, -68, -43, 1, 122, 0]
<9> load char* = &<8>
<10> call cJSON_AddFalseToObject (<4>, <9>)
<11> load int = 1
<12> load int = 0
<13> call cJSON_PrintBuffered ? (<4>, <11>, <12>)
```

Figure 3: Example program in the format of Hopper DSL.

pool. Second, Hopper selects inputs in the seed pool and mutates them with the guidance of coverage feedback.

Pilot Phase. In the pilot phase, Hopper draws skeletons of the inputs and infers constraints. Initially, the seed pool contains no inputs. Therefore, Hopper tries to generate simple inputs for each API function based on their signatures and learn constraints from them (detailed in Section 3.4). To accomplish this, Hopper selects an API function in the library as a target and attempts to randomly generate a call statement for it. This includes generating arguments and inserting related calls that introduce the necessary context. These statements finally form an input, which is then executed by the interpreter. If the input triggers a new path in the libraries without crashing, Hopper saves it into the seed pool for further mutation. To prevent irrelevant coverage feedback from other calls, Hopper tracks the coverage of the target call only.

Algorithm 1 describes the procedure of generating a call statement. Each argument is generated according to its type in the function signature using one of the following three operators:

- Hopper chooses an existing statement whose type matches the argument[1].
- The argument is obtained by inserting a new API invocation. Hopper randomly selects an API function whose return type matches the argument, and generates a call statement for it recursively. The new call statements are placed ahead of the current one. Additionally, an assertion statement indicating whether the call statement runs successfully is added after the new call statement (e.g., a non-null assertion statement is inserted after a pointer-type returned call statement).
- A load statement with a typed value created from scratch is used.

Hopper also attempts to affect the execution of the target call by inserting other API calls to change the internal states of the program. It prioritizes API functions with non-primitive argument types that overlap with those of the target call and reuses the overlapping arguments as much as possible. To prevent the program from becoming overly complex, Hopper stops generating new calls if the length of statements exceeds a specific threshold or the recursion depth becomes too high. Take the program in Figure 3 as an example. To generate the program with `cJSON_PrintBuffered` (Line 13), Hopper randomly chooses the return value of `cJSON_ParseWithOpts` (Line 4) and generate two integer value (Line 11 and 12) as its arguments. In addition, it creates a call for `cJSON_AddFalseToObject` that may modify existing arguments (Line 10).

[1]Hopper treats const and non-const types equally when comparing types.

Algorithm 1 Randomly generate a call statement based on its function signature. If the length of statements in the *program* exceeds a certain length, stop generating new calls recursively.

```
1:  function GenerateCall(program, sig)
2:      args ← empty list
3:      for all arg_type ∈ sig.arg_types do
4:          if Rand() < THRESHOLD then                ▷ Use existing statement as argument
5:              index ← Randomly choose a statement with arg_type in the program.
6:              if index exists then
7:                  args ← args ∪ index
8:                  continue
9:              end if
10:         end if
11:         if Rand() < THRESHOLD and arg_type is not primitive then    ▷ Generate a new call statement as argument
12:             f ← Randomly choose an API function that returns arg_type .
13:             if f exists then
14:                 call ← GenerateCall(program, f)
15:                 index ← InsertStmt(program, call)
16:                 check ← GenerateAssert(program, index)
17:                 InsertStmt(program, check)
18:                 args ← args ∪ index
19:                 continue
20:             end if
21:         end if
22:         load ← GenerateLoad(program, arg_type)    ▷ Generate a new load statement as argument
23:         index ← InsertStmt(program, load)
24:         args ← args ∪ index
25:     end for
26:     if Rand() < THRESHOLD then    ▷ Generate a new call statement that may affect the execution of call
27:         f ← Randomly choose an API function.    ▷ Prefer API functions whose argument types overlap with args
28:         call ← GenerateCall(program, f)
29:         InsertStmt(program, call)
30:     end if
31:     call ← CreateCallStmt(sig, args)
32:     return call
33: end function
```

Evolution Phase. After running a certain number of rounds, Hopper enters the evolution phase, which aims to build more complex programs based on the skeleton inputs. To achieve this, Hopper selects

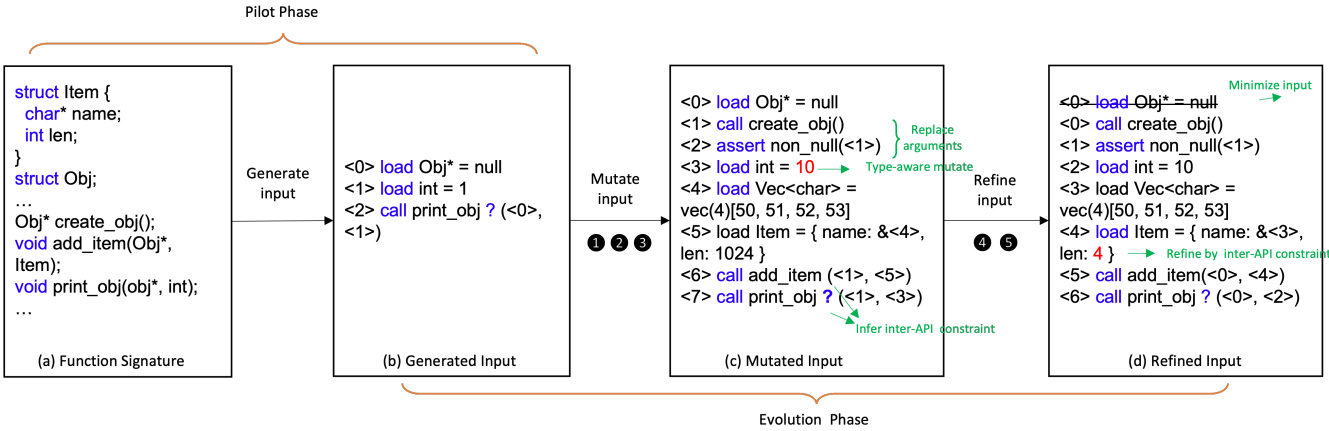

Figure 4: The workflow of input generation in Hopper.

an input from the seed pool and randomly mutates the statements based on their types. Guided by the execution branch coverage, Hopper keeps the mutated inputs that explore deeper code of libraries, which are more likely to find bugs. Overall, Hopper mutates program statements in five steps. ❶ Hopper selects an input from the seed pool with priority. Fresh seeds are given higher priority to be chosen, as they are more likely to reach deeper paths. ❷ Hopper chooses statements to mutate from the input according to their weights. The assert, file, and update statements are weighted as zero and not mutated, while the weights of load and call statements are determined by their complexity. ❸ Hopper mutates the load statements and call statements by their corresponding strategies, as described in Section 3.3.1 and Section 3.3.2 respectively. ❹ Hopper refines the input to use APIs correctly based on the constraints learned during fuzzing, as described in Section 3.4. ❺ Hopper minimizes the input to remove redundant statements that have no effect on reaching the path yet increase the search space of mutation, as described in Section 3.3.3.

3.3.1 Call Statement Mutation. Hopper adopts the following mutation strategies for call statements:

- Replace one of the arguments with a new argument that retains the same type (Line 4 to Line 24 in Algorithm 1).
- Insert a new call before the target call (Line 27 to Line 29 in Algorithm 1). The inserted call may modify the values of the target call's arguments or change the global states in libraries. Hopper determines the effectiveness of the inserted calls through branch feedback, which is detailed in Section 3.4.2.
- Update the return value of the call. An update statement is inserted after the call to partially overwrite the return value with a new value.

3.3.2 Type-aware Value Mutation. As libraries often use various argument types, including custom composite types, Hopper needs to generate appropriate types for such arguments when invoking the corresponding APIs. The same is true for value mutation, where applying mutations according to the value's type can more effectively explore new states. To achieve this, Hopper parses type definitions (e.g., `struct`, `enum` and `union`) and type aliases in header files recursively. It then generates new typed values using the following rules:

- **Primitive Types.** Almost all primitive types are numerical types. Thus, Hopper generates numbers within a small range of uniform distribution. Additionally, Hopper applies one of four mutations to the primitive values. (1) Set an interesting value (e.g., 0x80000000 for int); (2) Flip a bit or a byte; (3) Add or subtract a small number; (4) Set to a literal collected from a comparison instruction, if the value is used as one of the operands.
- **Array.** Based on the array's length and element type, Hopper generates a sequence of elements. If the length is variable, Hopper first randomly chooses a length. During mutation, Hopper selects one or more elements in the array and mutates them respectively. In addition, Hopper can resize the array by inserting or removing elements if its length is not fixed.
- **Structure.** Values with custom structure types are created by recursively generating their fields. When mutating custom structure values, Hopper randomly selects a field in the structure and mutates it according to its type.
- **Trivial Pointer.** Trivial pointers expose the layout of the pointer type (i.e., pointers to primitive data types and structure types). Hopper mutates them by the following operations: (1) Set to a null pointer; (2) Point to the location of an existing statement that holds the same pointer type; (3) Point to a newly generated array whose element type is the same as the pointee; (4) Point to the location of the value returned by a newly generated call statement. The call statement returns the same pointer type.
- **Nontrivial Pointer.** Nontrivial pointers, such as opaque pointers, void pointers, and function pointers, cannot be mutated straightforwardly due to their unpredictable nature. Hence, Hopper constructs them separately. For the opaque pointers, Hopper retrieves the API functions that initialize the pointers via either returning the pointers or filling them through reference. If void pointers have alias types, Hopper

treats them as opaque pointers as well. Otherwise, Hopper attempts to cast an arbitrary-length byte array to the required void pointer to see if it works (detailed in Section 3.4.1). For the function pointers, Hopper synthesizes empty functions with the required signatures at compiling stage to allow the fuzzer to use their addresses as pointers.

Besides, byte arrays may contain data with their own encoding, which is not defined in header files. For example, `cJSON_ParseWithOpt` in Figure 2 parses a byte array with JSON formatting. For such values, Hopper applies AFL's random mutations that are designed for byte arrays.

3.3.3 Input Minimization. Redundant statements and values slow down execution speed and increase search space during mutating, which makes fuzzing inefficient. To address this, Hopper employs two steps to minimize the inputs:

1. Minimize inputs after mutation and refinement. Hopper inspects the statement in the input backward, excluding the target call statement. If a statement is no longer referenced by other statements, Hopper deletes it. For example, in Figure 4, Line 0 in (c) is redundant after mutation and is thus removed by Hopper in (d).
2. Minimize inputs that trigger new paths. Hopper removes calls that have no impact on the execution path (detailed in Section 3.4.2), as well as redundant values in load statements. It tries to set the pointer values to null or shrink the length of arrays when possible. If the execution path remains unchanged, Hopper retains the mutation in the input.

3.4 Constraint Learning

To correctly invoke APIs, DSL programs generated by Hopper must satisfy both intra- and inter-API constraints. Intra-API constraints dictate that APIs must be invoked with appropriate arguments, while inter-API constraints specify the appropriate order in which APIs are invoked. Hopper learns these constraints through libraries' runtime feedback rather than relying on external sources.

3.4.1 Intra-API Constraint. We propose six general intra-API constraints based on our observations of real-world libraries.

- **Non-null Pointer(NON-NULL)**. APIs that do not check for null pointers can crash when invoked with null pointers. It's often unclear whether this is a real bug, as some developers argue it is the user's responsibility to perform null checks.
- **Valid File Resource(FILE)**. When an API function reads from or writes to a file, the file name provided as an argument must be valid. If the file name is randomly generated, the API call may terminate early, or it could mess up the disk if used as an output stream.
- **Specific Value(EQUAL)**. APIs that use a number in the arguments to designate the boundary of an array pointer may suffer from overflow errors if the number is incorrect.
- **Bounded Range(RANGE)**. APIs that access or allocate limited resources based on argument numbers may encounter resource exhaustion or overflow errors if the number is out of range.
- **Array Length(ARRAY-LEN)**. Some APIs assume that the arrays referenced by the pointers have sufficient elements rather than asking for arguments indicating boundaries. This may result in overflow errors if the arrays don't have enough elements.
- **Specific Type Cast(CAST)**. Due to missing layout information of the void type, developers have to generate objects with concrete types and cast their references to the void pointers.

Hopper learns intra-API constraints throughout the entire fuzzing process, including both the pilot and evolution phases. In addition to inferring constraints for the arguments themselves, Hopper recursively explores composite structures to infer constraints for any objects contained within, such as the pointees, fields in structs, and elements within arrays.

For `void*` arguments without alias types, Hopper assumes that their pointees do not contain any pointer. Therefore, they can be cast from a large enough random byte array, and Hopper adds CAST constraints that treat them as `char*` type. Conversely, in the case of `void*` pointers with alias types, Hopper supposes that they are opaque pointers and initializes them by invoking other APIs.

When new paths are explored by inputs, Hopper checks to see if any file open function (e.g., `fopen`) has been triggered, and compares the file name with the arguments used to invoke the API. If there is a match, a FILE constraint is created for the corresponding argument.

Furthermore, if an input triggers a new crash, Hopper infers intra-API constraints by the following steps in order.

(1) If the call triggers a segmentation fault due to accessing a null pointer (*si_addr* is 0 or close to 0, where *si_addr* is the address of the faulting memory reference), Hopper locates each null pointer in the arguments, sets it to the address of a protected memory chunk, and runs this mutated program again. If the program crashes again at the same program location (indicated by the RIP register) and triggers illegal access inside the protected memory chunk, it means the pointer is accessed without a null check in the API invocation. In that case, Hopper adds a NON-NULL constraint for this pointer.
(2) If the crash is caused by accessing a canary appended right after an array (*si_addr* is in the range of canary), Hopper tries to figure out whether there is a length or index of a variable-sized array in the arguments. Firstly, Hopper locates which array has been overflowed. We denote the array's length as N. For each numerical value in the call's arguments, Hopper attempts to set it as $N-1$, N, and $N+1$, respectively. If both N and $N+1$ lead to a crash by accessing the canary, Hopper adds a RANGE constraint to set the value within a range of $[0, N)$. If only $N+1$ makes a crash, an EQUAL constraint is added to set the value to be the same as the array's length.
(3) If the above strategies fail to rectify illegal access of a canary, Hopper attempts to pad the arrays in the arguments to a specific length K (e.g., 64) to see whether it resolves the crash. If so, an ARRAY-LEN constraint is added to ensure that this array is at least K bytes.
(4) For other illegal access, if there are CAST constraints for the arguments, Hopper tries to mutate the byte array pointed

to by the arguments. If the illegal address varies with the mutated bytes, the `void` pointer may be interpreted as a structure containing pointers. Thus, HOPPER removes the CAST constraints with `char*` type.

(5) If the inputs lead to a timeout or out of memory, HOPPER searches for large numerical values in the arguments and mutates them. If the execution becomes significantly faster or exits normally after setting the value to be small, HOPPER adds a RANGE constraint for the argument to limit its maximum value.

These constraints are used to refine inputs after mutation, and inputs that violate the constraints are excluded from the seed set. As an example, in Figure 4, HOPPER successfully infers the constraint that the value of `len` should be the length of the array that `name` refers to, and refines the value accordingly.

3.4.2 Inter-API Constraint. An API call can modify its referenced arguments or internal states in the program, which affects the behavior of its subsequent calls. These relationships between API calls are referred to as inter-API constraints. Compared to intra-API constraints, inter-API constraints are more difficult to identify and apply universally. Each library defines its own pattern to coordinate the API functions, making it hard to make abstractions. Instead of imposing concrete constraints, HOPPER preserves the inter-API constraints by saving the effective programs for later mutations. To learn such constraints, HOPPER statically assembles library APIs by function signatures and then verifies their effectiveness dynamically through coverage feedback.

Initially, HOPPER statically infers naive constraints between API functions by analyzing their signatures. By comparing the types of their arguments and returns, we can deduce whether the two API functions are likely related. For any given pair of API functions $\mathbf{F_1}$ and $\mathbf{F_2}$, they tend to be related if their types overlap in the following ways.

- Type $\mathbf{T}$ is the type of one of the arguments in $\mathbf{F_1}$, and it is the type of return value of $\mathbf{F_2}$.
- Both $\mathbf{F_1}$ and $\mathbf{F_2}$ use type $\mathbf{T}$ as their arguments.
- $\mathbf{F_1}$ uses an argument with type $\mathbf{T}$ while $\mathbf{F_2}$ uses a pointer $\mathbf{T}*$ that may modify the value it points to.

Moreover, if the identifiers of arguments in the latter two cases are identical, the two API functions are more likely to be related.

Two API functions have an effective relation if the former can help the latter to reach a new path. As shown in Algorithm 1, HOPPER attempts to generate a large number of various API usages. The API calls may be composited in an interesting order after several iterations. In case HOPPER finds a certain sequence of API calls to be effective, the program is reserved for further mutation. By precisely controlling coverage tracking, as described in Section 3.2.2, HOPPER learns the effective relations between API functions by the following steps.

(1) When new calls are inserted before the target API call, only the coverage of the target call will be tracked.

(2) If the insertions trigger a new path, HOPPER deletes the new calls one by one from the input and checks whether the coverage remains the same after each removal. If the coverage changes, HOPPER identifies the removed call as critical for reaching the new path. In other words, an efficient inter-API constraint between the two API functions is found in the program.

In addition to recognizing effective call sequences, HOPPER also identifies the effective arguments among sequences of API calls. Once a program triggers new paths, HOPPER checks whether the new paths were introduced by the mutation of certain arguments. To reuse effective arguments for further mutation, HOPPER caches all the statements that produce these effective arguments. These statements contain both concrete values and inter-API constraints. During mutation, HOPPER replaces the original argument with an effective argument from the cache to introduce the appropriate context to the API calls.

4 IMPLEMENTATION

We implemented HOPPER in 23164 lines of Rust code and 1285 lines of C/C++ code for instrumentation.

4.1 Fuzzer

HOPPER generates inputs with semantics extracted from C header files. Combining Rust's traits and macros, we implemented the process elegantly. First, we adopted Rust bindgen [30] to automatically generate Rust FFI bindings, including type definitions and function signatures, from library header files. For C++ libraries, HOPPER only accepts C-style API declarations since bindgen [16] does not yet support features like templates. Next, we defined `mutate`, `generate`, `serialize`, `deserialize` traits for the types in the bindings, which apply customized behavior to objects of each type for fuzzing. We manually implemented these traits for primitive types since they have common memory layouts in libraries. As for custom structures, we utilized Rust's *procedural macros* to automatically implement these traits. HOPPER also synthesizes empty functions for function pointer types according to their function signatures. When invoked as callbacks, these functions do nothing other than return an object with zero-initialization. Finally, we generated a table of object builders that HOPPER uses to call the trait implementations. These automatically generated codes are compiled and linked against the fuzzer.

To report the inputs that trigger bugs to developers for analysis, we have built a tool that translates DSL to C source code.

4.2 Interpreter

HOPPER instruments the library binaries using static binary rewriting with E9Patch [13]. We borrowed from E9AFL [15] the code for branch tracking and improved it to be controllable and API-sensitive. Moreover, the instrumentation collects the compare instructions and hooks resources management functions (e.g., `malloc`, `free` and `fopen`).

When executing inputs, HOPPER's interpreter parses them based on the code that implements `deserialize` traits. To invoke the APIs, HOPPER generates the code of the caller mapping table described in Section 3.2.2, using Rust FFI bindings and procedural macros. This code is then compiled into the interpreter. Similar to AFL, the interpreter uses the techniques of fork servers to reduce the overhead of process initiation. It forks a new process once it receives a new input for interpretation.

During interpretation, Hopper detects memory overflow by placing a page-size canary after the array values in a memory arena. This memory arena is implemented by `mmap` continuous memory at a fixed address. In the arena, the address of the last byte of an array is aligned to a page's last byte. To detect overflow, Hopper reserves a page size memory after the array and sets the page as unreadable and unwritable by `mprotect`.

5 EVALUATION

In this section, we evaluated Hopper on 11 widely-used real-world libraries. All of the selected libraries are commonly used by various applications and have been evaluated by OSS-Fuzz [31]. Their developers have crafted several fuzz drivers to adapt fuzzing on those libraries using LibFuzzer. Among all the 11 libraries, *re2* is distinct in that it is developed and exported as a C++ library. For the sake of compatibility, Hopper fuzzed *re2* through its C wrapper: *cre2* [12]. Additionally, all libraries included in our experiments were the latest versions available at the time of testing. To demonstrate the effectiveness of Hopper, the following research questions were answered:

- **RQ1:** How effective is Hopper in libraries?
- **RQ2:** Can Hopper correctly infer API constraints?
- **RQ3:** Can Hopper generate programs that are comparable with MCFs?

All of our experiments are conducted on a server with an Intel Xeon Platinum 8255C and 128 GB memory running 64-bit Ubuntu 20.04 LTS. In each experiment, we configured the tested fuzzer to be executed on only one core.

5.1 RQ1: How effective is Hopper in libraries?

We compared Hopper with MCFs and other automatic solutions (i.e., FuzzGen and GraphFuzz) on 11 widely-used real-world libraries, using code coverage and bug finding as metrics. For the MCFs, we selected fuzz drivers either crafted by the library developers or collected from the Google OSS-Fuzz project. The authors of FuzzGen provided the fuzz drivers for *libvpx* and *libaom*, which we used directly in our evaluation after rectifying the inaccuracies listed in Section A.2. However, FuzzGen encountered errors during fuzz driver generation, except for *cJSON*. These errors involved violating assertions while determining data types of arguments or getting stuck at static backward slicing. GraphFuzz requires manually written schemas to synthesize fuzz drivers. We adopted the author-provided schema for *sqlite3* and wrote the others ourselves. In each individual experiment, we ran the fuzzers with a timeout of 24 hours. Afterward, we recompiled the tested library with LLVM's source-based code coverage feature [32] enabled and re-ran the fuzzers with seed inputs to collect coverage information. To reduce statistical errors, we repeated each experiment five times and reported the average results. Additionally, for libraries containing multiple MCFs, we ran each driver with LibFuzzer for 24 hours separately.

5.1.1 Code Coverage. Table 2 shows the code coverage achieved by different fuzzers for each library. Hopper outperformed other fuzzers on all libraries in both lines and branches coverage, except for *re2*, *libvpx*, and *libaom*. Even though 9 MCFs of *zlib* have been

Table 2: Comparison of coverage between Hopper, Graphfuzz, FuzzGen, and MCFs on real world libraries.

Library	Fuzzer	Lines	Branchs
cJSON	cjson_read_fuzzer	952 (42.92%)	473 (46.56%)
	FuzzGen	186 (8.39%)	96 (9.45%)
	GraphFuzz	1346 (60.69%)	612 (60.24%)
	Hopper	1997 (90.04%)	854 (84.06%)
c-ares	parse_reply_fuzzer	1757 (23.94%)	744 (21.78%)
	create_query_fuzzer	110 (1.50%)	47 (1.38%)
	Total of MCFs	1865 (25.41%)	790 (23.13%)
	GraphFuzz	2424 (33.02%)	994 (29.10%)
	Hopper	5012 (67.06%)	2045 (59.03%)
libpng	libpng_read_fuzzer	5005 (28.67%)	2041 (26.42%)
	GraphFuzz	1629 (9.33%)	507 (6.56%)
	Hopper	9610 (55.04%)	3903 (50.53%)
lcms	IT8_load_fuzzer	641 (3.41%)	271 (3.21%)
	overwrite_transform_fuzzer	2964 (15.77%)	1271 (15.07%)
	transform_fuzzer	4382 (23.32%)	1757 (20.84%)
	Total of MCFs	5357 (28.50%)	2205 (26.15%)
	GraphFuzz	2481 (13.20%)	872 (10.34%)
	Hopper	9001 (47.89%)	3135 (37.18%)
libmagic	magic_fuzzer	3094 (30.87%)	2043 (28.69%)
	GraphFuzz	3197 (31.90%)	2003 (28.10%)
	Hopper	4230 (45.26%)	2634 (39.51%)
libpcap	fuzz_both	4301 (27.05%)	2346 (32.40%)
	fuzz_filter	5561 (34.97%)	2901 (40.07%)
	fuzz_pcap	974 (6.13%)	365 (5.04%)
	Total of MCFs	6870 (43.21%)	3479 (48.05%)
	GraphFuzz	1736 (10.92%)	881 (12.17%)
	Hopper	7536 (47.40%)	3669 (50.68%)
zlib	checksum_fuzzer	251 (5.21%)	61 (2.12%)
	compress_fuzzer	1880 (38.73%)	904 (31.48%)
	example_dict_fuzzer	1901 (39.17%)	952 (33.15%)
	example_flush_fuzzer	1631 (33.61%)	776 (27.02%)
	example_large_fuzzer	1684 (34.70%)	795 (27.68%)
	example_small_fuzzer	1429 (29.45%)	758 (26.39%)
	minigzip_fuzzer	2227 (45.89%)	1071 (37.29%)
	uncompress2_fuzzer	989 (20.37%)	468 (16.30%)
	uncompress_fuzzer	976 (20.11%)	457 (15.91%)
	Total of MCFs	2976 (61.32%)	1482 (51.60%)
	GraphFuzz	2602 (53.62%)	1343 (46.76%)
	Hopper	3502 (72.16%)	1914 (66.64%)
re2	re2_fuzzer	6373 (67.85%)	3367 (67.75%)
	GraphFuzz	5564 (59.23%)	2843 (57.21%)
	Hopper	6413 (68.27%)	3299 (66.38%)
sqlite3	oss_fuzz	22582 (30.96%)	9054 (25.99%)
	GraphFuzz	6261 (8.58%)	2172 (6.23%)
	Hopper	25356 (34.76%)	9551 (27.41%)
libvpx	vpx_dec_fuzzer_vp8	3604 (10.44%)	1138 (14.21%)
	vpx_dec_fuzzer_vp9	15654 (45.34%)	3475 (43.41%)
	Total of MCFs	18787 (54.42%)	4463 (55.75%)
	FuzzGen	15211 (44.06%)	3367 (42.05%)
	GraphFuzz	15060 (43.62%)	3251 (40.61%)
	Hopper	15641 (45.30%)	3514 (43.89%)
libaom	av1_dec_fuzzer	39762 (21.05%)	9722 (17.73%)
	FuzzGen	32576 (17.25%)	7757 (13.35%)
	GraphFuzz	30837 (16.33%)	7327 (12.61%)
	Hopper	36218 (19.18%)	9228 (15.88%)

tested for a total of 216 hours with Libfuzzer, Hopper achieved better results than the MCFs overall. Notably, Hopper generated inputs that covered almost three times as much code as the MCFs' best effort for *c-ares*. On the other hand, while the MCFs for *re2* were sophisticated enough to cover most of the source codes, Hopper managed to achieve a coverage level that is comparable to the MCFs. In the case of *libvpx* and *libaom*, MCFs covered more lines and branches than automatic solutions. This is because they can reach complicated internal states by decoding multiple frames in a loop, which is not possible for the automatic solutions that are loop-free. However, among the automatically generated fuzz drivers, Hopper explores more lines and branches than FuzzGen and GraphFuzz.

To demonstrate that Hopper is capable of fuzzing more APIs, we counted the number of unique APIs invoked by the valid programs generated by fuzzers. Table 1 shows that, on average, Hopper successfully invoked 93.52% of APIs, while MCFs, FuzzGen, and GraphFuzz only cover 18.58%, 13.93%, and 41.42% APIs respectively.

5.1.2 Bug Finding. For the crashes triggered by Hopper, we first eliminated any spurious crashes that violated the inter-API constraints learned by Hopper and then removed the duplicated crashes with the same stack traces at the program crash point. The remaining crashes were verified manually by inspecting the code, debugging the programs, and reading the official documents. Hopper ultimately discovered 25 new bugs from 51 unique crashes, as shown in Table 3. In contrast, none of the MCFs, FuzzGen, and GraphFuzz found a valid bug under the same 24 hours running. Of the 51 unique crashes, 26 were identified as spurious crashes as they violated the constraints specified in the documentation or were rejected by library developers. The developers did not classify these crashes as bugs since they have their own criteria for valid API usage. The APIs must be used strictly within specified constraints, which may be mentioned in the documentation, or the program may crash unexpectedly. For example, an out-of-bound read error could occur if the `zFunctionName` argument for `sqlite3_create_function16` is not a UTF-16 string [2].

We have reported all the discovered bugs to the corresponding communities. At the time of writing, 17 bugs have been confirmed by library developers and the rest are still under review. The details of the discovered bugs are listed in Table 4. Null pointer dereference is the most common type of bug detected by Hopper (11 out of 25). These bugs occurred when null pointers were produced or improperly initialized by library calls and accessed without null pointer checking within the code of API functions. Buffer overflow is another common type of bug detected (8 out of 25). Six of those overflows were detected by Hopper's canary when reading or writing out of bounds of arrays allocated as arguments by Hopper. The two remaining bugs involved accessing either uninitialized memory or the addresses of internally allocated arrays in the API functions beyond their bounds. Apart from these, Hopper identified three division by zero errors, an uncontrolled format string bug, a double-free bug, and an infinite loop, which are analyzed in detail in Section 5.3.

Most of the buggy APIs discovered by Hopper were not covered by MCFs, FuzzGen, or GraphFuzz. This is not surprising as the fuzz drivers for these libraries have been extensively utilized for continuous fuzzing over an extended period. To ensure a fair comparison with MCFs, GraphFuzz, and FuzzGen, we additionally evaluated Hopper and these fuzzers on previous versions of the 11 libraries. As shown in Table 5, Hopper outperformed MCFs, GraphFuzz, and FuzzGen by detecting 28 unique bugs in previous versions of the 11 libraries. Moreover, Hopper was able to identify 7 out of the 14 bugs that other fuzzers had detected. Although Hopper missed certain bugs reported by other fuzzers due to its inefficient mutation power on those predefined API calls, it prioritized mutating API usages that are more likely to trigger bugs or explore new states via its seed selection, which includes any API usage within the libraries. Consequently, Hopper discovered a greater number of bugs overall.

Table 3: Unique crashes reported by Hopper.

Library	Version	#Programs	Crashes				Accuracy
			#UC	#S	#B	#C	
cJSON	1.7.15	2,972	3	1	2	2	66.67%
c-ares	1.18.1	3,192	2	0	2	2	100%
libpng	1.6.37	5,612	8	4	4	1	50%
lcms	2.13.1	2,660	13	8	5	5	38.46%
libmagic	FILE5_42	1,662	0	0	0	0	-
libpcap	1.10.1	2,249	5	2	3	3	60%
zlib	1.2.12	5,598	4	1	3	3	75%
re2	0.4.0	27,355	4	2	2	0	50%
sqlite3	3.38.5	10,356	11	7	4	1	36.36%
libvpx	1.11.0	22,282	0	0	0	0	-
libaom	3.5.0	19,654	1	1	0	0	-
Total	-	**103,592**	**51**	**26**	**25**	**17**	**49.02%**

UC = Total unique crashes; **S** = Spurious crashes caused by incorrect API usage; **B** = Valid bugs that identified by manually review; **C** = Confirmed bugs after reported to library developers.

Answer to RQ1: Hopper outperformed MCFs and other library fuzzing approaches in terms of both code coverage and bug finding. Specifically, Hopper detects 25 new bugs and 17 of them have been confirmed.

5.2 RQ2: Can Hopper correctly infer API constraints?

One of the key insights that allows Hopper to generate correct and efficient API usages is its ability to learn intra- and inter-API constraints. To evaluate the benefits of these constraints, we compared Hopper with and without constraints on the 11 libraries. As shown in Table 6, Hopper with inferred constraints explores more lines and branches across all the libraries. However, without intra-API constraints, code coverage is substantially lower in *libpng*, *libmagic*, *libvpx*, and *libaom*. This is because certain API functions that many other APIs depend on could not be properly called. For instance, in *libpng*, `png_init_io` initializes a file input for a handler that is subsequently read by other APIs for processing. If the fuzzer randomly generates a string as the file's name without a FILE constraint, `png_init_io` will return an invalid handler, causing all subsequent calls to terminate at a shallow state. As incorrect API usages often lead to program crashes, we also

[2] https://sqlite.org/forum/forumpost/7ace1408b.

Table 4: Verified bugs found by Hopper.

ID	Library	Location	Buggy function	Bug Type	Status	Commit ID	MCF	FZ	GF
1.	cJSON	cJSON.c:2209	cJSON_DetachItemViaPointer	Null pointer dereference	Confirmed	722(P)	✗	✗	✗
2.	cJSON	cJSON.c:2326	cJSON_ReplaceItemViaPointer	Null pointer dereference	Fixed	726(P)	✗	✗	✗
3.	c-ares	ares_init.c:2254	ares_set_sortlist	Buffer overflow	Fixed	496(S)	✗	-	✗
4.	c-ares	ares_init.c:2247	ares_set_sortlist	Buffer overflow	Fixed	496(S)	✗	-	✗
5.	libpng	pngerror.c:229	png_warning	Buffer overflow	Confirmed	453(S)	✗	-	✗
6.	libpng	pngwrite.c:1976	png_image_write_to_file	Division by zero	Reported	489(S)	✗	-	✗
7.	libpng	pngwrite.c:1976	png_image_write_to_memory	Division by zero	Reported	489(S)	✗	-	✗
8.	libpng	pngwrite.c:1976	png_image_write_to_stdio	Division by zero	Reported	489(S)	✗	-	✗
9.	zlib	deflate.c:401	gzsetparams	Null pointer dereference	Fixed	761(P)	✗	-	✗
10.	zlib	gzread.c:517	gzungetc	Buffer overflow	Fixed	837(S)	✗	-	✗
11.	zlib	gzwrite.c:136	gzflush	Infinite loop	Fixed	840(S)	✗	-	✗
12.	re2	cre2.cpp:195	cre2_find_named_capturing_groups	Null pointer dereference	Reported	30(S)	✗	-	✗
13.	re2	cre2.cpp:725	cre2_set_match	Null pointer dereference	Reported	29(S)	✗	-	✗
14.	sqlite3	sqlite3.c:30121	sqlite3_overload_function	Uncontrolled format string	Fixed	bbbbb66b6b(T)	✗	-	✗
15.	sqlite3	sqlite3.c:79786	sqlite3_value_bytes	Null pointer dereference	Reported	0218d74c47(T)	✗	-	✗
16.	sqlite3	sqlite3.c:79786	sqlite3_value_bytes16	Null pointer dereference	Reported	0218d74c47(T)	✗	-	✗
17.	sqlite3	sqlite3.c:85266	sqlite3_value_subtype	Null pointer dereference	Reported	0218d74c47(T)	✗	-	✗
18.	lcms	cmsio1.c:857	cmsIsCLUT	Buffer overflow	Fixed	350(S)	✗	-	✗
19.	lcms	msgamma.c:852	cmsBuildTabulatedToneCurveFloat	Buffer overflow	Fixed	351(S)	✗	-	✗
20.	lcms	cmsnamed.c:760	cmsGetPostScriptCRD	Null pointer dereference	Fixed	353(S)	✗	-	✗
21.	lcms	cmslut.c:416	cmsStageAllocMatrix	Buffer overflow	Fixed	354(S)	✗	-	✗
22.	lcms	cmscgats.c:1928	cmsIT8SaveToMem	Buffer overflow	Fixed	355(S)	✗	-	✗
23.	libpcap	pcap.c:2946	pcap_breakloop	Null pointer dereference	Fixed	1147(P)	✗	-	✗
24.	libpcap	pcap.c:493	pcap_can_set_rfmon	Null pointer dereference	Fixed	1147(P)	✗	-	✗
25.	libpcap	pcap-linux.c:835	pcap_activate	Double Free	Fixed	1098(S)	✗	-	✗

Location : The source file location of the crash point. **Buggy function** : The API function that triggers the crash.
Commit ID : The bug trace id that was committed to community developers. **P**=Pull request number, **S**=Issue number, **T**=Bug forum id.
MCF, FZ, GF : Whether the bug could be discovered by MCFs, FuzzGen (FZ), and GraphFuzz (GF) under the same library version.

Table 5: Comparison with other fuzzers on previous versions of libraries.

Library	Version	MCF	GraphFuzz	FuzzGen	MCF ∪ GF	Hopper
cJSON	1.7.0	1	0	0	1	4(1)
c-ares	1.16.0	1	0	-	1	4(1)
libpng	1.6.32	0	0	-	0	4
lcms	2.8	2	1	-	2	6(1)
libmagic	FILE5_25	1	1	-	1	1(1)
libpcap	1.9.0	4	0	-	4	3(3)
zlib	1.2.11	0	0	-	0	1
sqlite3	3.22.0	1	0	-	1	3
libaom	1.0.0	3	1	0	4	2
Total	-	13	3	0	14	28(7)

Numbers in brackets show the number of bugs that were reported by multiple fuzzers including Hopper.

measured the success rate of execution for the generated inputs. Notably, after imposing intra-API constraints, *cJSON*, *c-ares*, *libvpx*, and *libaom* achieve almost 100% success rate of execution. In *re2*, the success rate improves significantly after Hopper learns not to generate null pointers as arguments. On the other hand, in the absence of inter-API constraints, Hopper explores API usage in a less efficient way by blindly combining API functions. This leads to poorer performance in terms of both line and branch coverage, particularly in libraries such as *libpng*, *lcms*, and *sqlite3*, which have a rich set of APIs available and therefore a large number of possible combinations to try out. To accelerate the search process, Hopper leverages inter-API constraints to pinpoint and reuse effective API usages.

Furthermore, we counted the number of intra-API constraints learned from the 11 libraries and analyzed their correctness. In total, Hopper learned 973 intra-API constraints. Among them, NON-NULL constraints were the most commonly inferred, as 609 out of the 973 constraints (62.01%) pertained to pointer dereferences without null checks in the API code. Additionally, Hopper was able to infer 17 FILE constraints, 147 EQAUL constraints, 35 RANGE constraints, 110 CAST constraints, and 55 ARRAY-LEN constraints across these libraries. The overall precision and recall of the learned constraints are 96.51% and 97.61%, respectively. Most of the false-positive constraints learned by Hopper are actually approximations of the ground truth constraints and work well in most cases. These constraints, although not completely correct, do help to reduce the invalid search space and prevent spurious crashes. For example,

(a) An uncontrolled format string bug found in *sqlite3*.

```
<0> load sqlite3 * = null //db
<1> load Vec<char> = vec(2)[96, 0] //file_buf
<2> file option <1> //filename
<3> load sqlite3 ** = &<0> //ppDb
<4> call relative: sqlite3_open ? (<2>, <3>)
<5> assert non_null(<0>
<6> load Vec<char> = vec(22)["JSFuMPRKm/RVAABApl6+
    vy6ib8NjAA=="] //zFuncName, decoded as "%!n0JU@^.o"
<7> load char* = &<6>
<8> load i32 = 55767322 //nArg
<9> call target: sqlite3_overload_function ? (<0>,<7>,<8>)
```

(b) A double free bug found in *libpcap*.

```
 <0> load char * = null //device
 <1> load char * = null //errbuf
 <2> call pcap_create ? (<0>, <1>) //p
 <3> assert non_null(<2>)
 <4> load pcap_t * = <2> //p
 <5> load int = 128 //buffer_size
 <6> call relative: pcap_set_buffer_size ? (<4>, <5>)
 <7> load int = 15 //snaplen
 <8> call relative: pcap_set_snaplen ? (<4>, <7>)
 <9> load int = 1833782204 //immediate
<10> call relative: pcap_set_immediate_mode ? (<4>, <9>)
<11> call target: pcap_activate ? (<4>)
```

Figure 5: Example programs that trigger bugs in *sqlite3* and *libpcap* respectively.

in `cmsStageAllocCLut16bitGranular`, setting the fourth argument `outputChan` to be the length of the fifth argument `table`, while not entirely accurate, worked effectively in most cases. Regarding the recall of the learned constraints, we identified false-negative constraints by analyzing the spurious crashes reported by Hopper since collecting the entire ground truth is extremely labor-consuming. Moreover, most of these missed constraints are specific to a particular library and hard to describe in a general way. Typically, a spurious crash indicates a violation of certain unlearned constraints. As an example from *sqlite3* described in Section 5.1.2, Hopper cannot ensure that the generated buffers are strictly UTF-16 encoded, which leads to a spurious crash.

Answer to RQ2: Hopper has the ability to learn intra-API constraints with high precision (96.51%) and recall (97.61%), while also being capable of speeding up the search process during fuzzing through the use of inter-API constraints.

5.3 RQ3: Can Hopper generate programs that are comparable with MCFs?

To demonstrate the ability of Hopper for generating high-quality fuzz drivers, in Figure 5, we select two representative programs generated by Hopper in our experiment as case studies.

Case 1. Figure 5a shows an uncontrolled format string bug found in *sqlite3* (ID 14 in Table 4). In `sqlite3_overload_function`, `zFuncName` is passed to `sqlite3_mprintf` to clone a string, but at worst, `sqlite3_mprintf` would perform formatting if the string contains specifiers. Hopper crafted this program successfully through the following steps. In the beginning, it generated a simple program that passes a null `sqlite3` pointer and a random `zFuncName` string to `sqlite3_overload_function`. However, since the API function requires a non-null `sqlite3`, Hopper used intra-API constraints to initialize the pointer by invoking `sqlite3_open`. This made the program reach the code that calls `sqlite3_mprintf`. Shortly thereafter, Hopper randomly mutates the strings for `zFuncName` and fortunately produces a '%' symbol in the string. This case indicates that Hopper is able to synthesize valid code to invoke an API function with intra-API constraints and find bugs using type-aware mutation for arguments.

Case 2. Figure 5b shows a double-free bug found in *libpcap* (ID 25 in Table 4). This bug occurs when `pcap_activate` fails to enable memory-mapped capture due to `mmap` with zero length under the immediate mode. In such a case, a memory location would be released twice in the error handling code. The length is computed from the combination of `buffer_size` and `snaplen`. Even worse, the `pcap_activate` is a commonly used API function for activating a packet capture handle rather than those rarely-used ones. *Tcpdump*, as a popular Linux application, is able to invoke those *libpcap* APIs sequentially by a simple command: `tcpdump -i any -B 1 -s 15 --immediate-mode`. By utilizing inter-API constraints, Hopper carefully crafted a sequence of API calls that triggered the bug. Specifically, it inferred that `pcap_set_buffer_size`, `pcap_set_snaplen`, and `pcap_set_immediate_mode` would modify `pcap_t` pointer to change the behavior of `pcap_activate`. Then, Hopper mutated the integers fed into the API calls and triggered the bug finally. The case demonstrates that Hopper can explore different API usages by learning inter-API constraints, which traditional fuzz testing tools typically ignore.

Answer to RQ3: With grammar-aware input fuzzing, Hopper synthesizes programs that satisfy both intra- and inter-API constraints. As compared to MCFs, the programs generated by Hopper can explore a much broader range of API usages.

6 DISCUSSIONS

6.1 Multiple-dimensional Search Space in Input Generation

Traditional library fuzzing generates a byte array as input and leaves the job of constructing arguments to fuzz drivers. In contrast, in Hopper, the search space for input generation is multidimensional as it involves both API functions and arguments, which poses a significant challenge. Furthermore, each argument has its own encoding format and requires specific mutation strategies. Although Hopper mitigates this issue by implementing novel techniques such as constraint learning and type-aware mutation, there is still much room for improvement.

Table 6: Impact of inter- and intra-API constraints on Hopper's efficiency.

Library	Learned intra-API Constraints										Success Rate		Line Coverage			Branch Coverage		
	#N	#F	#E	#R	#C	#A	TTL	#FN	PRC	RCL	w/o Intra	Hopper	w/o Intra	w/o Inter	Hopper	w/o Intra	w/o Inter	Hopper
cJSON	15	0	10	4	1	0	30	1	100%	96.77%	99.4291%	99.9999%	80.93%	88.86%	90.04%	75.00%	82.48%	84.06%
c-ares	87	0	26(1)	2	16(1)	2	133(2)	0	98.50%	100%	91.4087%	99.9999%	47.78%	65.31%	67.06%	43.91%	56.90%	59.63%
libpng	34(1)	2	20	5(1)	14	16(6)	91(8)	1	91.21%	98.81%	98.8973%	99.9917%	12.03%	39.05%	55.04%	7.39%	36.46%	50.53%
lcms	233	6	25(6)	1(1)	25(3)	13(3)	303(13)	8	95.71%	97.32%	95.3159%	99.2425%	46.65%	28.37%	47.89%	34.54%	21.05%	37.18%
libmagic	3	1	2(1)	0	3	1	10(1)	0	90.00%	100%	99.4833%	99.9935%	13.16%	42.20%	45.26%	9.96%	36.08%	39.51%
libpcap	59	4	1(1)	4	1	4(2)	73(3)	2	95.89%	97.22%	94.4091%	99.9797%	40.92%	40.89%	47.40%	43.56%	43.14%	50.68%
zlib	6	1	20	3	6	6	42	1	100%	97.67%	97.0134%	99.9865%	52.94%	59.28%	72.16%	44.46%	48.96%	66.64%
re2	93	0	26	0	0	4	123	2	100%	98.40%	86.1034%	99.9427%	58.21%	63.72%	68.27%	54.81%	61.91%	66.38%
sqlite3	72	3	8	10(2)	38(2)	9(1)	140(5)	7	96.43%	95.07%	99.3814%	99.5527%	30.52%	29.44%	34.76%	23.72%	23.34%	27.41%
libvpx	3	0	2	1(1)	4	0	10(1)	0	90.00%	100%	98.8642%	99.9999%	1.14%	42.51%	45.30%	3.35%	42.61%	43.89%
libaom	4	0	7	5(1)	2	0	18(1)	1	94.44%	94.44%	94.2413%	99.9999%	1.73%	16.67%	19.18%	3.12%	13.32%	15.88%
Total	**609(1)**	**17**	**147(9)**	**35(6)**	**110(6)**	**55(12)**	**973(34)**	**23**	**96.51%**	**97.61%**	**90.4769%**	**99.8808%**	**35.09%**	**46.94%**	**53.85%**	**31.26%**	**42.39**	**49.25%**

N = NON-NULL; **F** = FILE; **E** = EQUAL; **R** = RANGE; **C** = CAST; **A** = ARRAY-LEN; **TTL** = Total number; **FN** = False negative; **PRC** = Precision; **RCL** = Recall. Numbers in brackets show the number of false positive constraints.

6.2 Compatibility with C++ Libraries

Currently, Hopper only supports fuzzing libraries with C-style header files. The use of templates in C++ headers delays the compilation of template functions to the time of instantiation by the users, thus making it challenging for Hopper to generate callers of C++ functions and their arguments. Moreover, it is non-trivial to decide the concrete types of template parameters to instantiate the templates. To enable compatibility with C++, a more generalized implementation of generation and mutation is required. We plan to address this in future work.

6.3 False Positive Crashes

Although Hopper is highly effective at inferring common constraints to filter out most spurious crashes, the remaining crashes may still be false positives since the APIs need to be used in specific ways. Learning these constraints through dynamic feedback can be challenging as they have no universal criteria, as discussed in Section 5.2. However, during fuzzing, Hopper will no longer generate input for APIs that have failed to learn constraints and have a high probability of crashing spuriously. To make Hopper more practical and user-friendly, we plan to add warnings for users about unlearned constraints and provide a convenient way for them to add these custom constraints themselves.

7 RELATED WORK

7.1 API Fuzzing

APIs have long been favored as fuzzing targets, given their role in allowing code modules to interact with others. For example, RESTler generates requests for RESTful APIs using a grammar automatically inferred from Swagger specification and employs coarse-grained feedback from service responses to guide mutations to reach deeper service states [5]. Similarly, Pythia [4] and MINER [27] fuzz REST APIs using coverage-guided feedback and machine learning models. In the realm of kernel fuzzers, Syzkaller [38] is a coverage-guided fuzzer that generates system call sequences using system API descriptions. To further improve the efficiency and effectiveness of kernel fuzzing, MoonShine [28] generates seed inputs by extracting system call traces from the execution of real programs, while Healer [37] uses dynamic analysis to learn the relationships between system calls and uses that knowledge to guide input generation.

Besides the adoption of fuzzing in REST services and system APIs, OSS-Fuzz [31] gathers hundreds of manually written fuzzers to continuously fuzz APIs of open source libraries. In recent years, there has also been a trend towards generating fuzz drivers for library APIs automatically [6, 22, 45, 24, 18]. FUDGE [6] and FuzzGen [22] extract the code of API usage from practical code to create fuzz drivers. APICRAFT [45] and WINNIE [24] record the API call sequences from the execution trace of existing consumer programs and combine them to generate fuzz drivers. GraphFuzz [18] views fuzz drivers as dataflow graphs and performs graph-based mutation with a manually specified schema. UTOPIA [23] statically analyzes the library to identify attributes of API arguments and automatically synthesize valid fuzz drivers from existing unit tests. In contrast to generating fuzz drivers in limited domains, Hopper is a fuzzer for library APIs without any specific domain knowledge, including any knowledge from practical examples and schema specifications. Hopper searches the combinations of APIs and arguments in a large space and learns the features of high-quality inputs simultaneously through runtime feedback.

7.2 Grammar-aware Fuzzing

Grammar-aware fuzzers leverage grammar to generate structured inputs that bypass syntax checking at the beginning of the program execution [20, 39, 25, 46, 41, 3, 33]. For example, LangFuzz [20] and IFuzzer [39] utilize the syntax of JavaScript language to fuzz Javascript interpreters, while SQLsmith [33] and SQUIRREL [46] generate SQL queries for testing DBMSs based on SQL grammar. Superion [41] and NAUTILUS [3] improve grammar-aware fuzzing by combining code coverage guidance. These tools manipulate input as an AST and mutate it according to coverage feedback.

As writing grammar rules require much human effort, some fuzzers try to automatically learn the grammar [40, 17, 7, 42]. Skyfire [40] uses probabilistic modeling to learn grammar from inputs,

while Learn&Fuzz [17] employs a recurrent neural network model. Nevertheless, the accuracy of the learned grammar is dependent on the quality of the corpus provided. Alternatively, GRIMOIRE [7] automatically infers the structural properties of input language based on code coverage feedback. Similarly, Profuzzer [42] probes the types of input bytes through per-byte mutations.

General grammar-based fuzzers are intended to test parsers or interpreters by exploring all feasible combinations based on input grammar. However, Hopper operates differently by focusing on generating various effective API calls rather than exhaustively exploring the input language's grammar.

8 CONCLUSION

In this paper, we present Hopper, a novel fuzzer that aims to fuzz libraries without any domain knowledge required in crafting fuzz drivers. Hopper links the libraries under test against an interpreter, which takes DSL programs as input and drives libraries to perform requested fuzzing behavior. To generates effective API calls in the format of DSL, Hopper learns intra- and inter-API constraints in the libraries and mutates the inputs with grammar awareness. We evaluated the effectiveness of Hopper on 11 real-world libraries. Hopper outperformed MCFs and the other automatic solutions in both code coverage and bug finding. Specifically, Hopper found 25 new bugs that others could not find. Our experimental results demonstrate that Hopper effectively explores a vast range of API usages for library fuzzing out of the box.

9 ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. 1801751 and 1956364.

REFERENCES

[1] A fuzz driver generated by FuzzGen for libvpx. https://github.com/HexHive/FuzzGen/tree/master/examples/libvpx.
[2] A schema for generating sqlite3 fuzz drivers written by GraphFuzz. https://github.com/hgarrereyn/GraphFuzz/blob/master/experiments/sqlite3/in/f1/schema.yaml.
[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: fishing for deep bugs with grammars. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*.
[4] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498*.
[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: stateful REST API fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.
[6] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. 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*, 975–985.
[7] Tim Blazytko et al. 2019. GRIMOIRE: synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, 1985–2002.
[8] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. JIGSAW: efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1531–1531.
[9] Peng Chen and Hao Chen. 2018. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)*. San Francisco, CA, (May 2018).
[10] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: fuzzing deeply nested branches. In *ACM Conference on Computer and Communications Security (CCS)*. London, UK, (Nov. 2019).
[11] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz'em all: generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 642–658.
[12] cre2. https://github.com/marcomaggi/cre2/.
[13] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 151–163.
[14] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: data flow sensitive fuzzing. In *USENIX Security Symposium (USENIX Security)*. Boston, MA, (Aug. 2020).
[15] Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2021. Scalable fuzzing of program binaries with E9AFL. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1247–1251.
[16] Generating bindings to c++. https://rust-lang.github.io/rust-bindgen/cpp.html.
[17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.
[18] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 1070–1081.
[19] Samuel Groß. 2018. Fuzzil: coverage guided fuzzing for Javascript engines. *Department of Informatics, Karlsruhe Institute of Technology*.
[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, 445–458.
[21] How to split a fuzzer-generated input into several. https://github.com/google/fuzzing/blob/master/docs/split-inputs.md.
[22] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, 2271–2287.
[23] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2022. UTOPIA: automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 746–762.
[24] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. Winnie: fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*.
[25] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: a neural network language model-guided JavaScript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, 2613–2630.
[26] LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[27] Chenyang Lyu et al. 2023. MINER: a hybrid data-driven approach for REST API fuzzing. In *USENIX Security Symposium (USENIX Security)*. Ananeim, CA, (Aug. 2023).
[28] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, 729–743.
[29] Pulling JPEGs out of thin air. https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html.
[30] Rust-bindgen. https://github.com/rust-lang/rust-bindgen.
[31] Kostya Serebryany. 2017. OSS-Fuzz- Google's continuous fuzzing service for open source software. In USENIX Association.
[32] Source-based code coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.
[33] SQLsmith. https://github.com/anse1/sqlsmith.
[34] Prashast Srivastava and Mathias Payer. 2021. Gramatron: effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 244–256.
[35] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*.
[36] Structure-aware fuzzing with libFuzzer. https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.
[37] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 344–358.
[38] syzkaller - kernel fuzzer. https://github.com/google/syzkaller.
[39] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: an evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601.
[40] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[41] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.
[42] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. Profuzzer: on-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 769–786.
[43] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, 745–761.
[44] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.
[45] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: fuzz driver generation for closed-source SDK libraries. In *30th USENIX Security Symposium (USENIX Security 21)*, 2811–2828.
[46] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 955–970.

A APPENDIX

A.1 DSL Grammar

$$
\begin{aligned}
Program &:= Line \mid Line; Program \\
Line &:= Index\ Statement \\
Statement &:= Call \mid Load \mid File \mid Assert \mid Update \\
Call &:= \textbf{call}\ Name(ArgList) \\
Load &:= \textbf{load}\ Type = Value \\
Update &:= \textbf{update}\ Index[Fields] = Index \\
Assert &:= \textbf{assert}\ Rule \\
File &:= \textbf{file write} \mid \textbf{file read}\ Index \\
ArgList &:= Index \mid Index, ArgList \\
Rule &:= \textbf{non\_null}(Index) \mid \textbf{eq}(Index, Index) \\
Index &:= \text{<numeric literal>} \\
Name &:= \text{function name} \\
Type &:= \text{type name of value} \\
Value &:= \text{serialized value} \\
Fields &:= \text{path to locate a value inside struct or array}
\end{aligned}
$$

Figure 6: Grammar of Hopper's DSL.

As shown in Figure 6, an individual DSL Program consists of a sequence of lines. The index at the beginning of each line is a unique ID (e.g., an incremental number), which can be referenced by other statements. Since the values in arguments can have various types, our DSL encodes them accordingly. Primitive types are easy to convert to and from strings. In the case of composite data types, such as an array or custom struct, we serialize their internal elements, following a structured order similar to the JSON format. Specifically, when dealing with long lists of primitive values, we optimize them for efficiency by encoding them in Base64 (as seen in line 0 of Figure 3). However, serializing a pointer is tricky since it may point to a value shared by multiple objects. Therefore, we only serialize the destination to which it points in the DSL program, without including any additional information.

A.2 Examples of inaccurate fuzz drivers

```
415 struct vpx_codec_ctx *ctx_hEP_1;// = &ctx_hEP_0;
416
417 // Dependence family #3 Definition
418 dep_3 = (struct vpx_codec_ctx *)ctx_hEP_1;
419 // initializing argument 'cfg_Ywn'
420 struct vpx_codec_dec_cfg cfg_Ywn_0;
421
422 *(uint32_t*)((uint64_t)&cfg_Ywn_0 + 0) = (E.eat1() & 0x3f)
        + 1; /* UNKNOWN */
423 *(uint32_t*)((uint64_t)&cfg_Ywn_0 + 4) = 0; /* UNKNOWN */
424 *(uint32_t*)((uint64_t)&cfg_Ywn_0 + 8) = 0; /* UNKNOWN */
425 struct vpx_codec_dec_cfg *cfg_Ywn_1 = &cfg_Ywn_0;
426
427 if (vpx_codec_dec_init_ver(dep_3, dep_6, cfg_Ywn_1, 0, 12))
         { /* vertex #4 */
428    return 0;
429 }
```

Figure 7: An example of misuse of consumer code in *libvpx*'s fuzz driver [1] generated by FuzzGen.

Figure 7 shows a fuzz driver for *libvpx* generated automatically by FuzzGen. On Line 415, the value of `ctx_hEP_1` is an uninitialized pointer, which is then used directly as an argument in the call on Line 427. In addition, upon examining the fuzz driver for *libaom* that was released by the authors of FuzzGen, we found an incompatibility with the newest version of *libaom*. Specifically, the initialization of `aom_codec_dec_cfg` objects in the fuzz driver overwrites 17-20 bytes despite the size of the object being reduced to only 16 bytes.

The schema [2] for *sqlite3* is written by the authors of GraphFuzz. When the program fails to invoke `sqlite3_prepare_v2` on line 72, it calls `close_all` to release all `sqlite3*` pointers. However, other types of allocated resources, such as `sqlite3_str*` pointers, may still be leaked. Moreover, the schema does not verify whether `sqlite3*` is initialized successfully or not in `new_database`.