Automatic Library Fuzzing through API Relation Evolvement

Jiayi Lin*, Qingyu Zhang*, Junzhe Li*, Chenxin Sun*, Hao Zhou†, Changhua Luo*‡, Chenxiong Qian*‡
*The University of Hong Kong
Email: {linjy01, z1anqy, jzzzli, roniny}@connect.hku.hk, chluo@hku.hk, cqian@cs.hku.hk
†The Hong Kong Polytechnic University
Email: cshaoz@comp.polyu.edu.hk
‡Corresponding author

***Abstract*—Software libraries are foundational components in modern software ecosystems. Vulnerabilities within these libraries pose significant security threats. *Fuzzing* is a widely used technique for uncovering software vulnerabilities. However, its application to software libraries poses considerable challenges, necessitating carefully crafted drivers that reflect diverse yet correct API usages. Existing works on automatic library fuzzing either suffer from high false positives due to API misuse caused by arbitrarily generated API sequences, or fail to produce diverse API sequences by overly relying on existing code snippets that express restricted API usages, thus missing deeper API vulnerabilities.**

This work proposes NEXZZER, a new fuzzer that automatically detects vulnerabilities in libraries. NEXZZER employs a hybrid relation learning strategy to continuously infer and evolve API relations, incorporating a novel driver architecture to augment the testing coverage of libraries and facilitate deep vulnerability discovery. We evaluated NEXZZER across 18 libraries and the Google Fuzzer Test Suite. The results demonstrate its considerable advantages in code coverage and vulnerability-finding capabilities compared to prior works. NEXZZER can also automatically identify and filter out most API misuse crashes. Moreover, NEXZZER discovered 27 previously unknown vulnerabilities in well-tested libraries, including OpenSSL and libpcre2. At the time of writing, developers have confirmed 24 of them, and 9 were fixed because of our reports.

I. Introduction

Software libraries are essential components of the modern software ecosystem. They provide reusable code modules to facilitate software development, enabling developers to implement complex functionalities without reinventing the wheel. Complex libraries like OpenSSL expose thousands of Application Programming Interfaces (APIs), offering fundamental capabilities for software applications. However, vulnerabilities in these libraries introduce severe security risks [15], [29]. For example, a recent vulnerability in OpenSSL allowed remote code execution and could affect millions of infrastructure software systems [2].

Network and Distributed System Security (NDSS) Symposium 2025
24 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240750
www.ndss-symposium.org

To uncover vulnerabilities inside libraries, fuzzing is a widely used method. In the domain of library fuzzing, a crucial component that determines the overall fuzzing effectiveness is the *fuzzing driver*. A fuzzing driver bridges the fuzzing engine and the target library, providing input data to API arguments and establishing calling dependencies across library APIs. Human experts can study the target API usage and manually craft a fuzzing driver that properly constrains the input for testing a library [1], [5], [25]. However, this requires extensive effort to create and maintain the driver code, and it becomes ineffective due to the extensive and continuously developing APIs. In contrast to the manual approach, *automated library fuzzing* has become a de facto approach [16], [15], [19], [22], [26], [17], [21], [20], [29] to discovering vulnerabilities because it saves manual labor and can scale to extensive APIs.

However, automated library fuzzing has been a difficult research problem. In addition to the traditional fuzzing challenge of generating diverse inputs (*i.e.*, API call sequences) to uncover more crashes, there is a significant concern regarding how to simulate real-world API usage to discover library vulnerabilities, rather than merely triggering crashes caused by *API misuse*. While both API vulnerabilities and misuse can cause crashes, they have distinct causes and implications. Vulnerabilities arise from design or implementation flaws that developers need to address, whereas API misuse occurs when library users (or driver code) do not follow the correct programming patterns or intended usage. Library developers typically do not address issues caused by API misuse because they lack valid security implications.

The goal of library fuzzing is to generate *diverse* API sequences that *accurately* reflect libraries' real-world usage. This creates a dilemma: we aim to assemble API call sequences in diverse ways to find vulnerabilities, yet we must constrain the input spaces of these sequences to adhere to the correct API usage acknowledged by developers. To address this problem, a set of library fuzzing systems are proposed, and they can generally be categorized into two types. The first line of research trades diversity in fuzzing for accuracy. They *replicate* the sequence orders from existing code snippets (which we call API *consumers*) that invoke the target APIs [16], [22], [17], [29], [21], [26]. Consequently, they only mutate the API arguments. The other line of research

adopts more aggressive mutations [15], [19], [20] for diversity. These works can mutate API arguments and API invocation orders together to detect more crashes. While they potentially generate diverse and unseen API usage that uncovers more API vulnerabilities, they also suffer from many API misuse crashes. The overwhelming number of API misuse crashes, which require manual efforts to validate, cause developer fatigue.

This work presents NEXZZER, a system that achieves both high diversity and accuracy when generating API sequences for testing libraries. Regarding diversity, NEXZZER implements a novel driver architecture that is facilitated by an intermediate API description, *Liblang*, cooperating with a general interpreter. This modular architecture allows the fuzzing engine to freely arrange and constrain diverse API call sequences. Moreover, it opens a window for the fuzzer to dynamically adjust mutation strategies during fuzzing and gradually learns *implicit API relations*. Implicit API relations are common in software libraries. For example, an object deallocation API (e.g., *BN_free* in *OpenSSL*) should not precede other object consumption APIs, and an initialization API (e.g., *gmpz_init* in *libgmp*) should always precede others. Such relations are not explicit in the API prototype or apparent in static analysis of consumer code but could significantly impact fuzzing automation. While fuzzers can occasionally generate inputs encoding these implicit dependencies, without understanding these dependencies, the fuzzers cannot consistently generate high-quality API sequences. Indeed, our evaluation demonstrates that existing automatic library fuzzers cannot discover and adhere to these relations well, resulting in inevitable API misuse and heavy manual post-processing burdens, especially when testing large-scale APIs.

To extract implicit API relations, NEXZZER incorporates a novel *relation learning* stage. This stage works by tentatively mutating APIs to observe the execution state changes and reason the API relations from them. NEXZZER records APIs and their relations into a dynamic graph structure named *APIGraph*. During the learning stage, NEXZZER checks the execution state changes against the mutated API sequences. A transition between crashing and non-crashing execution, or vice versa, indicates the existence of implicit dependencies in the API call sequences. Thus, NEXZZER updates such relations into the APIGraph accordingly. In future iterations, the fuzzing strategies are constrained by the evolving relations encoded in the APIGraph. This allows NEXZZER to produce more accurate API calling sequences gradually.

Another contribution of NEXZZER is that it automatically filters out false positive crashes caused by API misuse, ensuring much more accurate results and practicability in automatic library testing. We empirically summarize rules to determine if a crash is caused by API misuse. Combined with the diverse API sequences, it allows for efficient vulnerability detection. Specifically, we identify two main types of API misuse: 1) unintended API argument values (e.g., invalid argument values specifying memory length), and 2) unintended API invocation orders (e.g., consuming resources after improper deallocation). For each type, we identify multiple rules associated with API misuse based on expert experience. For each triggered crash, NEXZZER automatically determines if it falls into these categories (and thus is API misuse) based on learned relations and applicable rules.

We conducted a comprehensive evaluation on NEXZZER to evaluate its effectiveness in detecting API vulnerabilities. We selected 18 representative libraries and the Google Fuzzer Test Suite [10] as our evaluation benchmark, and compared NEXZZER with three state-of-the-art automatic library fuzzers including UTopia [29], FuzzGen [15], and Hopper [20], and also manually-crafted drivers. In general, NEXZZER achieved significantly higher code coverage (by up to 48.78%) compared to other tools. Our ablation studies showed that the components in NEXZZER contributed to both coverage and vulnerability detection. We further compared the number of API vulnerabilities detected by different fuzzers and the efficiency of these fuzzers to automatically filter out API misuse. To this end, for each reported crash, we manually confirmed if it is misuse based on human expertise and feedback from library developers. The results of NEXZZER are promising: regarding new vulnerabilities, NEXZZER detected 27 new vulnerabilities in widely-used libraries like OpenSSL, with 24 confirmed by developers and 9 fixed at the time of writing; yet the sum of vulnerabilities detected by other tools was 8. Regarding the efficiency of filtering out API misuse, NEXZZER also performed better by automatically filtering out 93.96% API misuse crashes, compared to 28.65% in Hopper and 47.11% in UTopia.

In summary, we make the following contributions:

- We design a modular driver architecture, facilitated by an intermediate API description, to effectively scale library fuzzing automation to large targets with thousands of APIs.
- We design a hybrid API relation learning strategy on top of a dynamic structure APIGraph to model API usage and behavior patterns. We also propose an effective rule-based approach to automatically filter out API misuse.
- We implement the design in a prototype named NEXZZER. It was evaluated on 18 libraries and found 27 previously unknown vulnerabilities. The source code of NEXZZER is available at https://figshare.com/s/9539927ac84ee6a7ac14.

II. Background and Motivation

To automate library fuzzing, researchers proposed various methods [16], [15], [22], [21], [29], [20]. In this section, we illustrate the diverse API relations (§II-A), the differences between API misuse and vulnerabilities (§II-B), and existing works and limitations that motivate our work (§II-C).

A. API Relations

According to prior works [20], [29], the essential API usage for effective API fuzzing can be categorized into two types of relations: argument constraints within APIs (i.e., intra-API) and dependencies between APIs (i.e., inter-API).

Intra-API Constraints. The first aspect of correctly invoking APIs is using expected arguments. Fuzzing drivers

must follow the library's design and randomize arguments within constrained scopes to test the functionality without triggering false alarms. There are several types of intra-API constraints. First, arguments (*e.g.*, LENGTH arguments) constrained by other arguments within the same API should not be randomly mutated by the fuzzer. Second, constants like integers or strings denoting predefined behaviors do not require mutation for testing. Third. there are some arguments that the driver should constrain within proper ranges. For example, the argument representing the size of allocated memory should be properly mutated; otherwise, it could potentially exhaust CPU or memory resources.

Inter-API Dependencies. We categorize interdependent API relations into *control dependency* and *data dependency*. Data dependency indicates the *define-and-use* data flow between arguments and return values. For example, it is common for an API to use data returned from prior APIs as arguments. Control dependency indicates the requirement of control-flow order between certain APIs. For instance, in OpenSSL, *EVP_MAC_update* is designed to be preceded by calling *EVP_MAC_init* for initialization.

B. API Misuse

Based on the API relations described above, any invocation violating these constraints would cause **Undefined Behaviors** (**UB**). Although a UB may not always instantly crash the execution, the unrecoverable mistakes in the library's internal states are highly likely to crash further API invocation. Similar to most fuzzing methods [7], [5], we use execution *crash* to signal a UB during the fuzzing. The causes of UB are classified into two types in this work: **API misuse**, caused by incorrect API usage contradicting the intended library usage; and **API vulnerability** caused by mistakes made in the library implementation and should be fixed. Library fuzzing aims to avoid API misuse while identifying API vulnerabilities.

TABLE I: Comparsion of Different Automatic Library Fuzzers.

	Method	Usage Learning	Accuracy	Diversity
FUDGE	R	Consumer	◐	○
Utopia	R	Consumer (Unit Test)	●	○
Rubick	R	Consumer	●	○
APICraft	M	Execution Trace	○	◐
FuzzGen	M	Consumer	◐	◐
Hopper	M	Dynamic Learning	○	●
NEXZZER	M	Consumer + Dynamic Learning	●	●

R denotes methods that Replicate existing API sequence usage
M denotes methods that Modify API sequences

C. Existing Works and Limitations

Many works have been proposed for library fuzzing automation. In this subsection, we review existing works and discuss their limitations.

1) Existing Works: We categorized existing works into two types based on whether they choose to **R**eplicate exact API usage sequence from existing code or **M**odify API control flow orders. Table I shows the metrics we used to evaluate their performance, including accuracy and diversity. *Accuracy* refers to the ability of fuzzers to generate valid API usage instead of API misuse. *Diversity* refers to the ability of fuzzers to produce diverse API sequences for testing the target library. An ideal library fuzzer should produce diverse API sequences that accurately reflect valid API usage, i.e., it achieves both high accuracy and diversity.

Replicating Existing API Usage This line of research [16], [22], [17], [29], [21], [26] shares the same characteristic of preserving the original API invocation order and mutating only the argument values to test the APIs. They generate test code based on *consumers*, *i.e.*, the existing code snippets invoking the target APIs. For example, FUDGE [16] is one of the early works to replicate API usage by program AST slicing. Some following works [17], [21], [22] improve the static analysis for retrieving more accurate API usage. More recently, UTOPIA [29] steps further to inject random inputs to library unit tests, which are found to have fewer noises than other consumers.

```
1 BIGNUM *a = BN_new();
2 BIGNUM *b = BN_new();
3 BN_bntest_rand(a, 512,
4     0, 0);
5 BN_set_bit(a, i);
6 BN_copy(b, a);
7 BN_free(a);
8 BN_free(b);
```

Listing 1: Consumer 1

```
1 BIGNUM *a = BN_new();
2 BIGNUM *b = BN_new();
3 ...
4 BN_hex2bn(&a, "FFF...");
5 BN_mod_exp_mont_consttime(
6   c, a, b, n, ctx, mont);
7 BN_free(a);
8 BN_free(b);
```

Listing 2: Consumer 2

The way of replicating the existing API usage reduces the possibility of generating erroneous drivers. However, it depends on the quality and coverage of the consumers, which might not always be available or comprehensive. Take OpenSSL as an example, the API *BN_set_bit* and *BN_mod_exp_mont_consttime* only separately appear in two unit tests (simplified in Listing 1 and Listing 2). By connecting *BN_set_bit* with *BN_mod_exp_mont_consttime* together in a call sequence, we detected a previously unknown integer overflow vulnerability, which the developers later confirmed and fixed. However, this line of research could not detect this vulnerability because it separately tests the two APIs following their unit tests. Our work detects 27 new vulnerabilities because we are not limited to consumers like the unit tests. We thus conclude that these works have high **accuracy** but low **diversity** (the first three tools in Table I).

Mutating API Sequences To detect more vulnerabilities, several works [15], [19], [20] are proposed to mutate not only API arguments but also call sequences for covering wider library functionalities. For example, APICRAFT [19] profiles consumer execution traces and composes different API sequences based on data dependencies. FuzzGen [15] performs consumer coalescing that identifies common API calls between consumers and randomly schedules their different subsequent API calls. More recently, Hopper [20] proposes an interpretative driver to trigger diverse API call sequences with dynamic

intra-API constraint learning.

Although these works show improved **diversity** by mutating API call sequences, they face the **accuracy** challenge that leads to higher misuse. For instance, APICraft only analyzes execution traces of consumers and lacks crucial intra-API constraints (e.g., length parameters). FuzzGen analyzes consumer source code but does not consider API control dependencies, *e.g.*, the allocation and deallocation API relations. These two tools usually require fixing API misuse manually. To mitigate misuse, Hopper learns some intra-API constraints (*e.g.*, length value range). However, it uses simple type-matching heuristics between parameters and return types to build data dependencies, which is not sufficient. For example, Hopper cannot test APIs with field-sensitive [42] data dependencies between arguments, which are crucial in some software like *libavc*, *libhevc*, *etc.* Hopper also only considers simple inter-API relations by observing coverage changes, which is inadequate for legal sequence construction and API misuse identification. As a result, as will be shown in our evaluation, Hopper still has much misuse (71.35%) that requires extensive manual effort to filter out.

2) Challenges and Motivation: Existing works either detect fewer vulnerabilities by exactly replicating consumers or suffer from API misuse due to arbitrarily mutating API sequences, failing to achieve both high diversity and accuracy. We model this challenge into two parts: ***C1:*** *how to efficiently cover wide input spaces of extensive APIs*? ***C2:*** *how to accurately test APIs while avoiding API misuse crashes during fuzzing?* In the next section, we introduce the design of NEXZZER targeting these challenges.

III. System Design

We present NEXZZER, a system that automatically tests libraries with high accuracy and diversity. In this section, we discuss the components of NEXZZER.

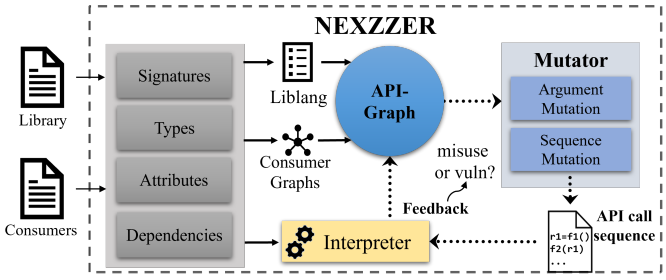

Fig. 1: Overview of NEXZZER.

A. Overview

The overview of NEXZZER is depicted in Figure 1. It first performs static analysis on the target libraries and consumer code to generate *Liblang* and a fuzzing *interpreter*. Liblang is an intermediate description that encodes basic API information like function signatures and argument types. NEXZZER parses Liblang to generate API call sequences, *i.e.*, Liblang seeds. The fuzzing *interpreter* receives Liblang seeds as inputs to invoke corresponding APIs. Together they form the basis of our *modular* driver architecture (§III-B), addressing **C1** mentioned in §II-C: the modular architecture decouples target-agnostic tasks from traditional driver code into the general interpreter, allowing for diverse API sequence scheduling and dynamic adjustment of API usage.

Diverse input space coverage naturally brings the risk of incorrect API usage and false positive crashes. To address this (**C2**), NEXZZER proposes a dynamic graph structure called APIGraph (§III-C) that encodes API relations. The APIGraph is initialized based on the consumer graph or type matching, depending on the availability of consumer code. Moreover, to discover new API usage beyond the consumer graph, NEXZZER dynamically evolves the APIGraph based on the execution feedback of the API sequences during fuzzing. The updated APIGraph, in turn, allows NEXZZER to adjust mutation strategies to produce more *accurate* API sequences. Finally, for API sequences leading to a crash, NEXZZER adopts a comprehensive rule-based approach to infer if they are caused by an API misuse or vulnerability (§III-D).

B. A Modular Driver Architecture

Our modular driver architecture consists of an intermediate description, *Liblang*, and a general interpreter. This approach differs from the one we call a *monolithic structure*, which amplifies much of the misuse in existing works. In this section, we first describe monolithic driver examples and their limitations, then illustrate our driver architecture.

1) A monolithic driver: A monolithic driver intermingles and completes several *preparation tasks* altogether, including structured input partitioning, argument value constraining, data dependency transferring, and call sequence scheduling, *etc.* Listing 3 is the monolithic driver produced by FuzzGen [15] to test OpenSSL by analyzing and coalescing the consumers in Listing 1 and Listing 2. It has an entry function `LLVMFuzzerTestOneInput()` that is invoked by the fuzzing engine with a random byte-array `data`. The driver first prepares argument dependencies (line 3). It then tests different API arguments in corresponding types with content partitioned from `data` (*e.g.*, `fuzzed_data.ConsumeString()`). It further randomizes API invocation orders for diversity (lines 6-28). To achieve this, FuzzGen generates *while-switch-case* loops (function pools), with each loop containing APIs selected from different consumers. Due to space limits, we show two function pools (lines 7-27) in the driver.

Automatically synthesizing the driver inevitably causes API misuse due to various reasons, such as over-approximation of static analysis. For example, the second argument value `bits` of `BN_bntest_rand()` in line 11 may vary in different consumers and thus be randomly mutated by the fuzzer, while it should be restricted to a proper range by the driver for fuzzing efficiency. Beyond argument usage, API sequence orders are also commonly misused. For instance, when FuzzGen coalesces consumers, it may inadvertently schedule the `BN_free()` API (line 21) before other dependent APIs, which results in typical Use-After-Free misuse.

```
1  int LLVMFuzzerTestOneInput(uint8_t *data, size_t size) {
2    FuzzedDataProvider fuzzed_data(data, size);
3    BIGNUM* r1 = BN_new();
4    ...
5    BN_hex2bn(&r1, fuzzed_data.ConsumeString());
6    ...
7    while (fuzzed_data.remaining_bytes() != 0) {
8      switch(fuzzed_data.ConsumeIntegral(0, NUM_API)) {
9      case 0:
10       int bits = fuzzed_data.ConsumeIntegral(0, RANGE);
11       BN_bntest_rand(r1, bits, ...);
12       break;
13     case 1:
14       BN_mod_exp_mont_consttime(r1, ...);
15       break;
16     }
17   }
18   while (fuzzed_data.remaining_bytes() != 0) {
19     switch(fuzzed_data.ConsumeIntegral(0, NUM_API)) {
20     case 0:
21       BN_free(r1);
22       break;
23     case 1:
24       BN_set_bit(r1, fuzzed_data.ConsumeIntegral());
25       break;
26     }
27   }
28   ... // other function pools
29 }
```

Listing 3: A Monolithic Driver Example

This inherent challenge is exacerbated in monolithic drivers. Because the preparation tasks are implemented using code and are intermingled with API usage in a monolithic way, one needs to carefully modify the driver code to avoid generating misuse. This is still a labor-intensive process that needs target-specific knowledge. For example, to avoid the misuse in Listing 3, one needs to restrict the `RANGE` (line 10) to proper values and constrain the invocation of `BN_free()` (line 21) to not precede other dependent APIs. Given the complexity of intra-/inter- API relations, no existing approaches attempt to automatically adjust their generated driver code. This makes the monolithic structure highly fragile, *i.e.*, whenever misuse occurs, the driver code might continuously produce API misuse crashes until manual post-processing fixes the misuse code.

2) Advantages of The Modular Driver: To address this challenge, we propose a modular driver architecture. Rather than completing all tasks within static driver code, the modular architecture decouples target-agnostic tasks from generating API call sequences. This enables NEXZZER to adjust generation strategies to avoid misuse dynamically.

Specifically, to produce and execute concrete API sequences, we implement an intermediate API description, *Liblang*, and a general *interpreter*. NEXZZER parses the API header files (*e.g.*, Listing 4) to obtain necessary information (*e.g.*, argument types) for generating *Liblang Descriptions* (*e.g.*, Listing 5). Our mutator (described in §III-D1) parses Liblang Descriptions and generates/mutates *Liblang seeds*. Each Liblang seed contains a concrete API call sequence (*e.g.*, Listing 6). We implement an interpreter to receive a Liblang seed as input and call different APIs based on the seed. To adjust generation strategies, NEXZZER does not modify any code. Instead, it constrains the mutator when mutating the

```
1  int DriverEntry(uint8_t *raw_api_seq) {
2    APISeq api_seq = Interpret_and_setup(api_seq);
3    for (int i = 0; i < api_seq.len(); i++) {
4      Transfer_dependencies(api_seq);
5      switch (api_seq.apis[i].idx) {
6      // -------- Synthesized code starts --------
7        case 0:
8          BN_set_bit(api_seq.apis[i].args[0], ...);
9          break;
10       case 1:
11         dep_1 = BN_new();
12         ...
13     // -------- Synthesized code ends ----------
14     }
15   }
16   Collect_feedback(api_seq);
17 }
```

Listing 7: NEXZZER's Interpeter

Liblang seed. For instance, to avoid Use-After-Free misuse in Listing 3, NEXZZER will not add `BN_free()` before its dependent APIs in Liblang seeds.

Liblang. Liblang is an automatically generated declarative description that abstracts essential API information. It is generated from function declarations and type definitions, consisting of API names, parameter/return types, and necessary structures' definitions. To automatically generate Liblang descriptions, NEXZZER takes library header files that declare *exposed* APIs, parses them into Abstract Syntax Trees (ASTs), and scans all the API declaration nodes. NEXZZER distinguishes exposed APIs from internal APIs using the function symbols' visibility in compiled binaries. After obtaining the target API set, it recursively scans all definitions of non-primitive types used by API parameters/return values and transforms them following the Liblang format.

Listing 4 and Listing 5 are examples of a library header and its generated Liblang. The syntax of Liblang is inherited from Syzlang [11] used for describing kernel system calls in Syzkaller [14], since system calls have similar signatures to library function calls. However, to suit the scenario of user-space libraries, we improve the syntax with extra types and utilities. The first one is the pointer resource type (*ptrres* in Listing 5). In Syzlang, system calls cannot return dependent pointers, while in Liblang, we extend the type system to support API-returned or user-created pointers as a dependent argument between library APIs. The second type is the support for file operations that are common in user-space libraries. Moreover, Liblang types are designed to be dynamically adjustable for API relation learning during fuzzing. For example, the `len[s]` (*i.e.*, the length of the first argument *s*) type in line 10, Listing 5 is originally an *int32* type and dynamically updated to a *len* type during learning (§III-D3).

Interpreter. We depict the structure of our *interpreter* in Listing 7. It calls different APIs based on the Liblang seed (`raw_api_seq` in line 1). The interpreter first parses `raw_api_seq` and completes the *preparation tasks*, such as setting up memory layouts (line 2) for non-primitive arguments and transfer-

```
typedef struct bignum_st BIGNUM;
BIGNUM* BN_new();
int BN_set_bit(BIGNUM* a, int n);
int BN_add(BIGNUM* a,
    BIGNUM* b,
    BIGNUM* n
);
BIGNUM* BN_bin2bn(
    const unsigned char* s,
    int len,
    BIGNUM* ret
);
...
```

Listing 4: A Library Header Example

```
bignum_st { placeholder int32 }
BN_new() ptrres[out, bignum_st]
BN_set_bit(a ptrres[in, bignum_st],
    n int32)
BN_add(a ptrres[in, bignum_st],
    b ptrres[in, bignum_st]
    n ptrres[in, bignum_st]
)
BN_bin2bn(s ptr[in, string],
    len len[s],
    ret ptrres[inout, bignum_st]
) ptrres[out, bignum_st]
...
```

Listing 5: A Liblang Description Example

```
resource0 = BN_new()
resource1 = BN_new()
resource2 = BN_bin2bn(
    &(0x20000040="\x0c\x01\x03",
    3, 0)
BN_add(resource0,
    resource1,
    resource2)
BN_add(resource0,
    resource1,
    resource2)
BN_set_bit(resource1,
    0x7fffffff)
```

Listing 6: A Liblang Seed Example

ring dependent arguments across API invocations (line 4). To test different libraries, the interpreter consists of *synthesized code* (lines 5-11) that includes all the API entries. Given a Liblang seed, the interpreter invokes corresponding APIs with the provided arguments (*e.g.*, `api_seq.apis[i].args[0]` in line 8) in the order specified by the seed (line 5). In this way, the preparation tasks are decoupled from the target-specific synthesized code.

Our interpreter also collects execution feedback at the end of executing an API sequence (line 16), including return status, code coverage, error information [8], *etc.* Such feedback is used by NEXZZER to dynamically learn implicit API relationships, as described below.

C. APIGraph

Different from some works that purely use *static* consumers to replicate API call sequences, we construct `APIGraph` to learn and express valid API usage *dynamically*. APIGraph is initially constructed based on the consumer graph or type matching, depending on the availability of consumer code. In this section, we introduce its basic structure and initialization. We will discuss the dynamic evolution of APIGraph in §III-D

1) APIGraph Structure: APIGraph is a directed graph, consisting of nodes and edges as described below.

Node. A node corresponds to an API function, defined by the API name and the types of its parameters. It has the following attributes:

▷ **Parameters**, which encompass the parameter types and attributes.

▷ **Constraints**, which are categorized into two kinds: *Value-Set* constraints and *Dependency* constraints. Value-Set constraints refer to specific parameter attributes (i.e., CONSTANT, RANGE, RANDOM, LENGTH, or FILEIO) with corresponding values (*e.g.*, constant or range of values). Dependency constraints indicate the presence of APIGraph edges on this node.

▷ **SeedSpace**, which stores *key-value* pairs where each *key* is a de-duplicated libLang seed that crashes at this node, and the *value* is corresponding API relations learned from the *key*. It also stores extra information such as execution feedback that serves for the de-duplication (§III-D3).

Edge. APIGraph edges are connected to parameters within the nodes when data dependencies are involved, or connected to a whole node for control flows. There are four edge types:

▷ **Control-Flow Edge**, which indicates control-flow edges between APIs appeared in consumer graphs.

▷ **Def-Use Edge**, which represents an explicit data dependency, where a target API node uses an argument defined by the return value or argument of a source API node.

▷ **Valid Edge**, which represents an implicit data dependency that the presence of the source API node is necessary for executing the target API node. For example, in OpenSSL, there is a Valid Edge from the API *EVP_MAC_init* to the API *EVP_MAC_update*, because the latter API can not function correctly without calling the former initialization API.

▷ **Invalid Edge**, which represents an implicit data dependency that the presence of the source API node would lead to unintended usage or undefined behavior of the target API node. For example, in OpenSSL Big Number APIs, an Invalid Edge exists between *BN_free* to *BN_add*, because calling the former API before the latter one leads to Use-After-Free misuse.

Note that the Valid and Invalid Edges are implicit dependencies and cannot be accurately determined by static analysis. Instead, NEXZZER learns implicit dependencies during the fuzzing process. APIGraph accordingly evolves with new Valid/Invalid Edges to reflect discovered dependencies.

2) APIGraph Initialization: There are two ways to initiate an APIGraph, depending on the availability of consumers (*e.g.*, unit testing code).

Utilizing Consumer Graphs. If consumers are available, we utilize consumer graphs to initiate APIGraph. NEXZZER scans each existing function that consumes at least one API call to generate a consumer graph, where nodes represent API function calls and edges represent their control flows and data flows in consumers. Similar to FuzzGen [15], NEXZZER assigns different attributes to API arguments by performing backward intra-procedural data-flow analysis on each API argument: **FILEIO** is assigned to the arguments with data-flow reaching file-related functions (*e.g.*, `fopen()`) or the argument having file name patterns (*e.g.*, `"filename"`); **LENGTH** is assigned to arguments with data-flow reaching related built-in functions, *e.g.*, `API(s, strlen(s))`. **CONSTANT** denotes encountering constant values; **DEPENDENT** is assigned to

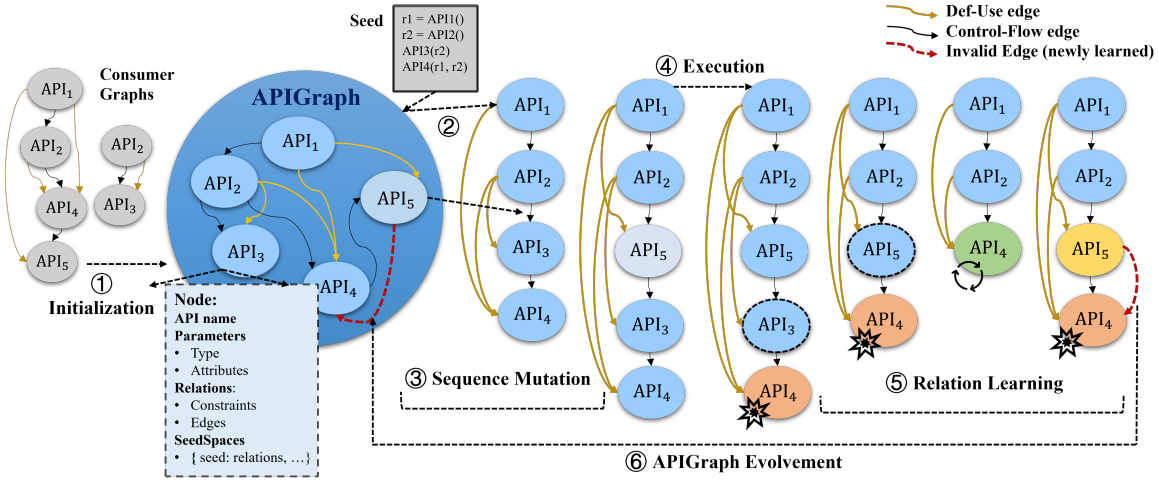

Fig. 2: APIGraph and an example of a fuzzing iteration.

the arguments with data-flow that overlap with another API, which forms Def-Use edges; and **RANDOM** is assigned to the rest arguments. When encountering pointer and aggregate argument types during data-flow analysis, we perform Anderson's points-to analysis [43] and track pointer dereferences. The analysis retrieves field-sensitive attributes for aggregate types by continuously slicing the data-flow of subfields.

Similar to previous works [15], our static analysis has certain limitations. For instance, it does not account for more implicit aliases or correlated constraints, where one argument value depends on another. Additionally, we cannot precisely handle function pointers and follow previous practices [15], [20] to manually adjust them if they are crucial. We consider these inherent limitations of static analysis to be orthogonal to this work and leave them as a future work.

Upon constructing consumer graphs, we merge consumer graph nodes that share the same API names and parameter types into one APIGraph node, conservatively retaining all API attributes. Edges are also merged based on the source, target, and edge type.

Type Matching. When there are no consumers, the APIGraph is initialized by matching parameters and return types to build Def-Use edges. The involved parameters are assigned with a **DEPENDENT** attribute. We also assign the **FILEIO** attribute to the arguments with file name patterns, and the **RANDOM** attribute to all rest arguments. Note that type matching may introduce over-approximated Def-Use edges. We discuss this with an example in §V-D2.

D. API Fuzzing

This subsection describes NEXZZER's fuzzing iterations. We provide the pseudocode algorithm of a fuzzing iteration in NEXZZER in Appendix §C3. At the high level, constrained by APIGraph, the mutator generates Liblang seeds. After executing the mutated seed, NEXZZER infers and updates the intra-API and inter-API relations in APIGraph based on the execution feedback. Based on the updated APIGraph, NEXZZER adjusts mutation strategies (*e.g.*, avoiding the deletion of out-edge nodes of Valid Edges). For each triggered crash, based on its learned relations and execution feedback, we use a rule-based approach to infer if it is a vulnerability or misuse.

1) Generating Liblang Seeds: NEXZZER designs two approaches for Liblang seed generation and *randomly* chooses one for each fuzzing iteration: one based on the consumer graphs (if available) and the other based on APIGraph. Generating seeds from consumer graphs can reproduce high-quality call sequences. To this end, NEXZZER starts from the root node in a consumer graph and performs a depth-first-search along the Control-Flow edges, generating multiple sequential paths leading to any leaf node. Intuitively, each generated path is a potential execution trace of the consumer.

On the other hand, seeds generated from APIGraph facilitate a wider exploration of input spaces. To produce a Liblang seed from APIGraph, NEXZZER randomly selects a self-contained node (*i.e.*, not a target of any Def-Use or Valid Edge) from APIGraph and then undergoes multiple rounds of *node insertions* (described in §III-D2). This allows NEXZZER to produce inputs expressing diverse API usages beyond consumers.

2) Seed Mutation: In this step, NEXZZER randomly triggers either the type-aware argument mutation or sequence-aware mutation on Liblang seeds.

▷ **Type-Aware Argument Mutation** targets independent arguments and applies different strategies tailored to their Liblang types and attributes: for arguments with **LENGTH** attribute, NEXZZER obtains the desired length values based on the corresponding pointer argument; to identify which pointer the **LENGTH** argument pertains to, NEXZZER mutates **LENGTH** argument to cause overflow and examine the access violation address [8]; for **FILEIO**, NEXZZER creates temporary files with mutable content; for arguments with value range constraints or **CONSTANT** attributes, NEXZZER performs constrained mutation within ranges; for arguments with pointer or structure types, NEXZZER recursively iterates into their pointee or member types for mutation when the field-

sensitive attributes are available (§III-C2); for arguments with **RANDOM** or without attribute, NEXZZER randomly mutates their values.

▷ **Sequence-Aware Mutation** targets **DEPENDENT** arguments between APIs. It involves randomly mixed rounds of two basic operations: *node insertion* and *node deletion*.

During node insertion, the mutator selects a random position within the Liblang seed and calculates potential candidate APIs that can be inserted. The algorithm is provided in the Appendix algorithm 1. Specifically, the insertion should not introduce Invalid Edges and should maintain all known Def-Use and Valid Edges for existing arguments and nodes.

Node deletion is an operation used not only for mutating seeds but also for relation learning in §III-D3. Given a Liblang seed and a candidate node to be deleted, the mutator searches for the candidate's Def-Use and Valid out-edges, identifying nodes that depend on the candidate node. The mutator would either remove the candidate and its dependency nodes, or retain the candidate node and search for another candidate.

3) API Relation Learning: NEXZZER discovers new API relations based on the feedback of executing Liblang seeds. It accordingly updates the APIGraph by adding new edges or updating API parameter attributes.

Selecting Seeds for Relation Learning. Not all mutated Liblang seeds are valuable for API relation learning. NEXZZER only analyzes the *de-duplicated* seeds whose executions lead to new crashes or new code coverage. NEXZZER deduplicates crashing seeds based on whether they have unique call stack frames of library's source code path (excluding unrelated paths like the interpreter/Glibc). While this approach may treat crashes with different call stacks as unique even if they are duplicates, we adopt this approach to avoid missing crashing seeds that indicate critical API usage during relation learning. We leave better deduplication as a future work. NEXZZER then performs the following learning phase to infer usage constraints.

Learning Relations Across APIs. NEXZZER infers new relations across different APIs (*i.e.*, nodes in APIGraph). It accomplishes this by examining whether executing an API can change the status of executing the API sequences. Specifically, for a crashing node N_t, if removing the node N_s (which has no known edge from/to N_t in APIGraph) changes crashing executions to non-crashing at N_t, it indicates that N_s and N_t have implicit dependencies that are not currently represented in APIGraph. In such cases, new Invalid Edges will be added to reflect these dependencies. On the contrary, for a non-crashing N_t, if removing N_s changes the execution to crashing, a new Valid Edge will be added.

We use the example in Figure 2 to illustrate this process. In a fuzzing iteration, a seed randomly selected from the corpus has the sequence: $API_1 \rightarrow API_2 \rightarrow API_3 \rightarrow API_4$. During seed mutation (§III-D2), NEXZZER inserts API_5 into this sequence and executes the new seed. Assuming the execution leads to a new crash of API_4, NEXZZER creates a new SeedSpace in the API_4 node and starts to analyze this sequence. It starts from the last node before API_4 and tries to remove each node *from back to forth*. It focuses on API_3 and API_5 because these two nodes have no relation with the crashing node API_4 according to the current APIGraph. If removing API_3 causes the execution to crash with the same feedback, NEXZZER removes API_3 to *minimize* the seed. If removing API_5 causes the execution to non-crashing, NEXZZER adds a new **Invalid** Edge from API_5 to API_4. In the following Liblang seed mutations, NEXZZER will not add API_5 before API_4 to avoid repetitive crashes. Algorithm 2 in the Appendix shows more details, including the rationale of the *back to forth* iteration.

Learning Value-Set Constraints of API Arguments. In addition to implicit API relations, NEXZZER also learn Value-Set constraints of API arguments and store them in the SeedSpace. Specifically, when encountering crashing APIs, NEXZZER tentatively adjusts argument values based on the execution feedback. For arguments that have "mutable" types (*e.g.*, integer types, pointers to primitive types, etc.) or attributes labeled with **RANDOM** and without **DEPENDENT**, NEXZZER tentatively changes the argument values using the following strategies: 1) For integers, NEXZZER performs binary searching of the value. 2) For other types of arguments, NEXZZER borrows values from prior non-crashing API sequences and consumers. By observing the execution results of APIs with mutated arguments, NEXZZER records a range of argument values that change the crashing status. This allows NEXZZER to avoid repeatedly triggering similar crashes due to incorrect usage.

4) Distinguishing API Misuse from Vulnerabilities: For each crashing seed, NEXZZER uses its learned relations (*e.g.*, Value-Set constraints) to identify if it is misuse. Specifically, we apply any applicable rules as summarized below to identify and filter out misuse. Our filtering rules only target crashing API misuse. Although NEXZZER might produce non-crashing misuse seeds, they will not have much impact as NEXZZER is coverage-guided. We also do not perform filtering for crashes caused by tentatively mutating seed during relation learning. These derived crashes are used as a reference to infer the original seed's API usage.

We summarize two types of misuse and their filtering rules: misuse caused by unintended 1) API argument values or 2) API invocation orders. For the first type, we consider the following cases. ① *Length:* A crash at the API is caused by a memory overflow, but it can be resolved by properly calculating an argument value indicating correct memory size (as described in §III-D2). ② *Range:* A crash is caused by CPU/memory exhaustion (indicated by execution time-out or out-of-memory error), but it can be resolved using a restricted argument value, such as the size argument for a memory allocation API. ③ *Non-null:* A crash caused by a pointer dereference of an invalid value (*e.g.*, NULL). This occurs when NEXZZER provides an invalid value to a pointer argument and observes an illegal dereference of the value.

The other type of API misuse is caused by unintended API call sequences. This includes the following cases. ① *Miss-*

DU: A crash occurs in an API sequence that contains API N, but the sequence does not include APIs that connect to N via a **Def-Use** Edge. ② *Invalid Usage:* A memory error (*e.g.*, UAF) occurs but the API sequence contains nodes connected by an **Invalid** Edge. Meanwhile, N_s contains memory deallocation according to the crashing call stack, *i.e.*, a user-created UAF misuse.

After applying these rules, crashes that are still unidentified would be assigned the *unknown type as suspicious vulnerabilities* for manual filtering. In our evaluation, through manually analyzing the crashes to obtain ground truth, we find that our rule-based filtering approach is effective, *i.e.*, it automatically filters out most (93.96% of) API misuse while retaining all true vulnerabilities. We also provide more examples and discuss the rationale of this approach in Appendix §B.

IV. System Implementation

We implemented the modular driver architecture with 1K lines of Python and 2K lines of C++ code based on the LLVM project [39], [41]. The fuzzing engine and dynamic learning strategies are implemented with 12K lines of Rust atop the LibAFL [23] framework, with a Liblang parser extended from Healer [38] with additional types (§III-B2). We discuss some implementation details in this section:

C/C++ Support. Similar to Hopper [20], our implementation supports testing C APIs, including C++ libraries that implement C-style APIs. However, unlike Hopper, which calls C APIs through Rust FFI in the fuzzing engine, NEXZZER synthesizes a standalone interpreter with API call entries in C (*e.g.*, lines 5-11 in Listing 7). This modular design is more easily extendable to other languages, such as C++ or other targets with callable interfaces. This is because Rust FFI is inherently challenging to use with interfaces other than C, even for C++ [9]. To extend NEXZZER's driver to support C++ APIs, it requires further engineering work to support class object transferring, class method invocations, and other common C++ grammars, which we leave as future work.

APIGraph Partitioning. A large library and its APIGraph usually consist of multiple independent groups of nodes (*i.e.*, some APIs are completely unrelated), which suggests separate testing to maintain efficiency. To this end, we apply classical graph partitioning algorithms to reduce the graph size. We first divide the graph into Weakly Connected Components (WCCs) [46] by performing breadth-first-search through its edges. As the initial APIGraph connects APIs through Control-Flow and Def-Use edges, a WCC ideally represents a group of APIs with related usage. If a WCC remains large, we further apply the Louvain community detection algorithm [37] to partition it into smaller groups of more closely related APIs. The algorithm takes a WCC as input, and we treat all edges in the WCC as equally weighted. After experimenting with several thresholds, we found that 1,000 is suitable to balance the size and the number of partitioned graphs. In our evaluation, we partition the APIgraph when it has over 1,000 nodes. We acknowledge that there may be relationships between APIs in different groups. We defer the precise categorization of APIs for large libraries to future work.

V. Evaluation

In this section, we comprehensively evaluate NEXZZER to answer the following research questions.

- **RQ1**: How effective is NEXZZER in testing library APIs compared to prior works?
- **RQ2**: Can NEXZZER effectively filter out API misuse?
- **RQ3**: Is NEXZZER practical and how much human effort is needed to validate its results?

A. Setup

We first describe our experimental setup, including the fuzzers we are comparing to, the target libraries, and the experimental settings.

Fuzzers. We compare NEXZZER with three state-of-the-art automatic library fuzzers named FuzzGen [15], Hopper [20], and UTopia [29]. We also compare with manually written drivers [1]. We cannot compare with automatic fuzzers such as FUDGE [16], Intelligen [17], and DAISY [21] as they have not open-sourced their implementations. Additionally, some works target different platforms, such as RUBICK [22] for Java libraries, APICRAFT [19] for macOS, and WINNIE [26] for Windows, and thus cannot be used for our evaluation. GraphFuzz [18] also targets API sequence testing, while it is not automated and requires manually-written schemas.

Evaluation Datasets. We selected two datasets for our evaluation. The first dataset includes 18 software libraries in their latest version. Our selection criteria include: 1) libraries that are evaluated in related works like UTopia [29] and Hopper [20]; 2) libraries that are widely used and heavily tested, such as OpenSSL and libxml2; The second dataset is the Google Fuzzer Test Suite (FTS) [10]. We test different fuzzers on this benchmark to evaluate their effectiveness in detecting known vulnerabilities.

Experiment Setting. We conducted all experiments using the same hardware configuration: a server with 64GB DRAM and 128 AMD EPYC 7702P processor cores at 3.3GHz. Each fuzzing driver was run for five rounds, with each round lasting 24 hours on a single CPU core. Following other works [15], [29], we excluded static analysis (which took less than 20 minutes on average) from the 24-hour time budget.

To compute library code coverage, we used SanitizerCoverage [40] in LLVM and averaged the results of the five rounds. NEXZZER's interpreter records code coverage of non-crashing executions. When a fuzzer has multiple drivers for a single library, we aggregated their coverage to calculate the overall code coverage of the library for that fuzzer. We used an empty initial corpus for all the experiments.

B. Effectiveness of NEXZZER (RQ1)

In this section, we evaluate NEXZZER and compare it with other tools.

TABLE II: Number of covered APIs (#UAPI), code coverage and standard deviation, and number of detected unique vulnerabilities (#UVul) of different fuzzers.

	Fuzzers	OpenSSL	tesseract	jsonnet	leveldb	uriparser	libvpx	libaom	libTom	libGMP	relic	libxml2	libpcre2	lcms	cJSON	libopux	libgsm	libavc	libhevc
#UAPI	BASELINE	606	9	5	15	17	17	13	33	59	54	59	14	10	6	4	5	1	1
	FUZZGEN	N/A	N/A	N/A	N/A	N/A	5	7	N/A	N/A	N/A	N/A	N/A	N/A	N/A	12	7	1	1
	UTOPIA	N/A	356	6	91	24	43	109	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	HOPPER	3834	138	46	68	90	24	35	93	594	270	114	27	274	78	54	36	1	1
	NEXZZER	3834	138	46	68	90	24	35	93	594	270	114	27	274	78	54	36	1	1
Code Coverage	BASELINE	13215(4%)	17013(9%)	1408(6%)	2247(8%)	7351(4%)	**2592(5%)**	**8329(8%)**	776(4%)	2817(6%)	2388(3%)	9541(8%)	**7737(4%)**	2222(3%)	621(1%)	1332(2%)	213(1%)	**4525(4%)**	**5658(4%)**
	FUZZGEN	N/A	N/A	N/A	N/A	N/A	1378(5%)	6028(6%)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1033(1%)	209(3%)	4119(3%)	3574(3%)
	UTOPIA	N/A	3951(2%)	1825(4%)	**2668(3%)**	7465(2%)	1728(6%)	7047(4%)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	HOPPER	17989(15%)	19400(10%)	2037(11%)	1000(10%)	7312(5%)	1461(8%)	7202(9%)	1142(4%)	5872(4%)	3490(6%)	5718(10%)	6544(10%)	1701(2%)	863(2%)	**4418(6%)**	312(0%)	33(0%)	21(0%)
	NEXZZER	**26984(9%)**	**20315(12%)**	**3434(5%)**	1877(9%)	**7835(4%)**	2467(13%)	7381(13%)	**1326(6%)**	**6450(4%)**	**4192(5%)**	**14195(9%)**	6825(7%)	**3306(3%)**	**865(2%)**	4021(4%)	**324(0%)**	4202(5%)	4733(4%)
#UVul	BASELINE	0	0	0	0	0	0	0	0	0	**2**	0	0	0	0	0	0	0	0
	FUZZGEN	N/A	N/A	N/A	N/A	N/A	0	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0	0	0	0
	UTOPIA	N/A	1	0	0	0	0	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	HOPPER	2	0	0	0	0	0	0	0	0	0	0	1	1	**1**	0	0	0	0
	NEXZZER	**18**	**1**	0	0	0	0	0	**1**	**1**	**2**	**1**	**1**	**1**	0	**1**	0	0	0

1) Evaluation Results of Libraries: Table II shows the testing results on the latest versions of 18 libraries. We found that FuzzGen and UTopia were unable to test certain libraries (marked as N/A in Table II). The reason for UTopia is that it requires unit tests written in particular unit test frameworks such as gtest [6] or boost [3] to test library APIs. However, not all libraries are integrated with such frameworks. For FuzzGen, its limited practicability, including missing header inclusions, incorrect type casts, incorrect dereferences, *etc.*, has been discussed by prior works [29]. Despite our efforts to fix some implementation issues, we still found it difficult to address them all. Therefore, we did not compare with FuzzGen for certain libraries where it is not applicable.

API Coverage. We count the number of unique APIs in different fuzzers' drivers. We found that NEXZZER and Hopper achieved higher API coverage in most libraries compared to other fuzzers. This is because NEXZZER and Hopper extract the target APIs directly from library header files, whereas UTopia and FuzzGen can only test APIs that appear in consumer code. UTopia achieved higher API coverage than NEXZZER in four C++ libraries because UTopia tests C++ APIs while NEXZZER only tests their partial C wrappers.

Code Coverage. NEXZZER achieved the best code coverage in 11 out of 18 cases. We provide an ablation study on NEXZZER regarding code coverage in §V-B3. The manually written drivers delivered a better performance for some small sets of target APIs (e.g., *libpcre2*). This is because manually written drivers can carefully craft an API sequence that sufficiently covers the major input spaces of the library, whereas NEXZZER allocates additional resources to sequence mutation. Hopper, in general, lacks more precise API usage learning to reach deeper coverage. Notably, it shows exceptionally shallow coverage in *libavc* and *libhevc* due to missing crucial field-sensitive data dependencies between the structure members of arguments. UTopia delivered better results when the C++ APIs and unit tests were comprehensive (e.g., *leveldb*).

Vulnerability Detection. We further compare the number of API vulnerabilities detected by different fuzzers. To this end, we manually analyzed all triggered crashes (excluding API misuse crashes indicated by fuzzers) to determine if they were real vulnerabilities and responsibly reported them to developers for confirmation.

From the results, NEXZZER detected significantly more vulnerabilities than other tools in the latest versions of libraries: it detected 27 new vulnerabilities (listed in Table V in Appendix), whereas the sum of vulnerabilities detected by other tools was eight. Most (24/27) vulnerabilities have been confirmed by the library developers, and nine were fixed at the time of writing. Eight confirmed vulnerabilities in OpenSSL have not been fixed because they involved deprecated APIs.

We investigated the reasons that NEXZZER detected more vulnerabilities than other tools. Compared with the baseline and UTopia [29], NEXZZER produced diverse API call sequences without being limited to the usage patterns in unit tests. Compared to Hopper [20], NEXZZER avoided producing erroneous sequences by learning implicit dependencies. This allowed it to assign more energy to test API sequences reflecting correct usages. In Appendix §A, we showcase two new vulnerabilities uniquely detected by NEXZZER in OpenSSL and PCRE2, both of which have been fixed by the developers.

2) Evaluation Results of Benchmark: We also evaluated known vulnerability detection because prior works evaluated these old versions of libraries. To this end, we tested the Google Fuzzer Test Suite (FTS) [10] to evaluate NEXZZER and other fuzzers. The reasons for not including all libraries in FTS include: (1) some are C++ libraries without C APIs; (2) some libraries are used to assess coverage without showcasing known vulnerabilities; We compared NEXZZER with Hopper [20] and the provided manual drivers. We did not include UTopia [29] because there is little overlap of targets that UTopia can run to compare with NEXZZER due to UTopia's reliance on particular unit test frameworks.

Table VI in the Appendix shows the number of uncovered known vulnerabilities of different libraries and Figure 4 shows the Venn diagram of the vulnerabilities. NEXZZER detected more vulnerabilities than other approaches in the FTS dataset. The reason that NEXZZER did not detect four vulnerabilities detected by manually crafted drivers was that these vulnerabilities did not lead to execution crashes (*e.g.*, logical errors requiring extra *assertions* manually written in the drivers), which could be one of the future direction of NEXZZER.

3) Ablation Study: To evaluate the effectiveness of individual components in NEXZZER, we conducted an ablation study to evaluate: **1):** the contribution of static consumer analysis; **2):** the contribution of dynamically learned relations. Other key components, such as the modular driver architecture, are

TABLE III: Consumer Analysis Results and Ablation Study

Targets		Consumer Results and Ablation								Relation Ablation						
		Consumer				Coverage			ΔBug	Coverage			FP Ratio			ΔBug
		#F	#N	#E*	#E	w/o C	NEXZZER	ΔCov		w/o R	NEXZZER	ΔCov	w/o R	NEXZZER	ΔFP	
OpenSSL	**P1**	296	568	16607	20852	9408	10904	**+15.90%**	2	7784	10904	**+28.61%**	0.2583	0.0377	**-85.40%**	1
	P2	547	951	11773	17346	11967	12358	**+3.27%**	0	9267	12358	**+25.01%**	0.1965	0.0289	**-85.29%**	0
	P3	255	634	2917	7642	12506	15076	**+20.55%**	0	9796	15076	**+35.02%**	0.1884	0.0254	**-86.51%**	0
	P4	262	625	4341	6638	10357	13983	**+35.01%**	0	9575	13983	**+31.52%**	0.1188	0.0309	**-73.98%**	1
	P5	117	423	3390	4892	9652	10352	**+7.25%**	0	9414	10352	**+9.06%**	0.1985	0.0122	**-93.85%**	0
	P6	24	254	794	1603	4987	5869	**+17.69%**	1	5397	5869	**+8.04%**	0.0416	0.0085	**-79.56%**	1
	P7	200	539	1439	3078	9011	9149	**+1.53%**	0	8537	9149	**+6.68%**	0.1188	0.0208	**-82.49%**	0
libpcre2		40	78	165	834	6511	6825	**+4.82%**	1	6171	7266	**+15.07%**	0.0512	0.0069	**-86.52%**	1
tesseract†		N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	16925	20315	**+20.03%**	0.2018	0.0679	**-66.35%**	2
jsonnet		6	48	106	141	3091	3434	**+11.10%**	1	2410	3434	**+42.49%**	0.1313	0.0127	**-90.33%**	0
leveldb		87	92	8	204	1810	1877	**+3.70%**	0	1612	1877	**+16.44%**	0.0152	0.0075	**-50.66%**	0
uriparser		3	70	71	179	7313	7835	**+7.14%**	0	6637	7835	**+18.05%**	0.1341	0.0881	**-34.30%**	0
libxml2		21	1182	16255	22972	13489	14195	**+5.23%**	1	10072	14195	**+28.86%**	0.1246	0.0079	**-93.65%**	0
cJSON		19	79	2459	2763	865	865	0.00%	0	686	865	**+20.69%**	0.1837	0.0237	**-87.09%**	0
lcms		4	287	3382	4120	3011	3306	**+9.80%**	2	2649	3306	**+19.87%**	0.1138	0.0187	**-83.56%**	2
libTom		19	180	3872	5640	1298	1374	**+5.86%**	0	948	1374	**+31.00%**	0.1045	0.0722	**-30.90%**	1
Relic		59	361	420	2538	3877	4192	**+8.12%**	0	3296	4192	**+22.08%**	0.2795	0.0348	**-87.54%**	0
libGMP		87	594	1339	7495	6019	6450	**+7.16%**	0	3201	6450	**+50.37%**	0.2205	0.0568	**-74.24%**	0
libopus		4	67	49	245	2965	3276	**+10.49%**	0	2197	3276	**+32.93%**	0.2249	0.0933	**-58.51%**	0
libvpx		4	37	108	218	2290	2467	**+7.73%**	0	1940	2467	**+19.40%**	0.0001	0.0001	**0.00%**	0
libaom		4	35	113	201	6889	7381	**+7.14%**	0	5642	7381	**+23.56%**	0.0001	0.0001	**0.00%**	0
libgsm		4	35	16	35	318	324	**+2.99%**	0	292	301	**+2.99%**	0.0001	0.0001	**0.00%**	0
libavc		2	53	0	280	33	4202	**+12633%**	0	3438	4202	**+18.18%**	0.0001	0.0001	**0.00%**	0
libhevc		2	29	0	135	46	4733	**+10189%**	0	4371	4733	**+7.64%**	0.0001	0.0001	**0.00%**	0

#F: the number of consumer files; **#N**: the number of (possibly duplicated) API call nodes in consumers; **#E***: the number of Def-Use Edges by type-matching; **#E**: the number of all edges after consumer analysis;
w/o C: running NEXZZER without the results from consumer analysis (i.e., only use type-matching edges); **w/o R**: running NEXZZER without the relation learning to constrain the mutator;
†: We did not find consumers using the C APIs of *tesseract* in its repository.

essential for running the fuzzer and cannot be ablated. The resulting differences in code coverage and the number of uncovered vulnerabilities are reported as **ΔCov** and **ΔBug**, respectively, in the **Consumer Results and Ablation** and **Relation Ablation** columns in Table III.

Consumer Analysis We ran NEXZZER on libraries with and without API usage from consumer analysis. In the latter case (**w/o C**), we did not assign attributes (from static analysis) to APIGraph nodes or generate seeds from consumer graphs.

When enabling consumer analysis, NEXZZER demonstrated about 10% improvements in code coverage across most targets, except for two targets (libavc and libhevc), which showed significant gains (about 12000%) with consumer analysis. The exception is because static analysis provides crucial API usage patterns that are only apparent in consumer code, which involve type casting and argument dependencies within structures. In terms of uncovered vulnerabilities, NEXZZER identified 8 more vulnerabilities when enabling consumer analysis. Overall, the results indicate that NEXZZER performs well without consumer data in most cases, but in certain libraries, consumer analysis is necessary.

Relation Learning We disabled the relation learning component in NEXZZER and retain the same APIGraph to investigate the benefits of relation learning. We find that relation learning consistently improved coverage across *all* cases, with gains ranging from 2.99% to 50.37%. In terms of uncovered vulnerabilities, NEXZZER detected 9 more vulnerabilities due to relation learning. The coverage and vulnerability number improvements can be attributed to fewer API misuse crashes in NEXZZER, as shown in **ΔFP** column under **FP Ratio** in the **Relation Ablation** section. We discuss and provide insight into how relation learning helps avoid API misuse in §V-C3.

C. Filtering out API Misuse (RQ2)

This subsection discusses the correctness and effectiveness of our relation learning and rule-based approach.

1) Correctness: NEXZZER triggered 7,291 crashes in Table IV, among which it automatically filtered out 6,851 and left 440 crashes as suspicious vulnerabilities. Through manual analysis of these 440 crashes, we confirmed 27 vulnerabilities. One might wonder if there were vulnerabilities among the 6,851 crashes automatically filtered out by NEXZZER. To evaluate this, we further manually analyzed all the 6,851 crashes. We explain how we analyze crashes and the time taken for the analysis in §V-D. We confirmed that all 6851 API misuse identified by NEXZZER were true cases of API misuse, indicating that NEXZZER could filter out API misuse with high correctness.

2) Effectiveness: In this part, we evaluate the effectiveness of different fuzzers in filtering out API misuse. While all tools in our evaluation could filter out certain API misuse, they all have false positives in identifying API misuse crashes. We thus study the crashes *not* tagged as API misuse by fuzzers to evaluate if they were unrecognized API misuse. We list the results in Table IV. In *OpenSSL*, due to its large-scale of APIs, we partitioned it into seven APIGraphs to fuzz separately as described in §IV. The partitioning is deterministic across the five rounds of fuzzing.

TABLE IV: Filtering out API Misuse and Relation Learning Effectiveness

Targets		NEXZZER										HOPPER				UTopia			
		ValueSet				APIEdge			Unknown Crash			Total	#Inf	#FP	#TP (BUG)	Total	#FP (Driver)	#TP (Driver)	#Bug
		Total	#L	#N	#R	Total	#Mis-DU	#I	Total	#FP	#TP (BUG)								
OpenSSL	P1	236	8	219	9	217	209	8	30	28	2	806	110	696	0	N/A	N/A	N/A	N/A
	P2	255	11	223	21	286	268	18	31	30	1	828	335	493	0	N/A	N/A	N/A	N/A
	P3	236	8	218	10	313	303	10	36	35	1	134	31	103	0	N/A	N/A	N/A	N/A
	P4	203	2	196	5	237	225	12	26	23	3	230	60	170	0	N/A	N/A	N/A	N/A
	P5	210	12	195	3	227	203	24	29	28	1	40	24	16	0	N/A	N/A	N/A	N/A
	P6	84	0	75	9	95	72	23	14	13	1	70	38	32	0	N/A	N/A	N/A	N/A
	P7	233	35	182	16	198	169	29	44	42	9	300	81	217	2	N/A	N/A	N/A	N/A
tesseract		95	6	89	0	47	37	10	29	27	1	86	9	76	1	246	111	135	0
jsonnet		29	0	29	0	16	7	9	4	3	0	22	4	18	0	15	0	15	0
leveldb		53	10	41	2	57	48	9	8	8	0	36	23	13	0	157	71	86	0
uriparser		8	3	3	2	60	49	11	7	7	0	93	46	47	0	80	61	19	0
libvpx		48	0	43	5	59	54	5	2	2	0	9	2	7	0	17	14	3	0
libaom		90	**5**	69	16	30	25	5	15	15	0	10	4	6	0	109	37	72	0
cJSON		73	16	57	0	39	39	0	7	6	0	11	9	1	1	N/A	N/A	N/A	N/A
lcms		206	8	198	0	261	260	1	12	9	1	106	2	103	1	N/A	N/A	N/A	N/A
libTom		271	14	176	81	345	320	25	13	12	1	194	106	88	0	N/A	N/A	N/A	N/A
Relic		260	33	161	66	443	396	47	43	41	2	2	2	0	0	N/A	N/A	N/A	N/A
libGMP		397	117	227	53	320	309	11	41	40	1	737	227	510	0	N/A	N/A	N/A	N/A
libopus		74	5	62	7	38	36	2	11	10	1	125	21	103	1	N/A	N/A	N/A	N/A
libpcre2		101	8	92	1	77	71	6	6	4	1	213	30	182	1	N/A	N/A	N/A	N/A
libxml2		95	14	78	3	169	161	8	31	30	1	11	0	11	0	N/A	N/A	N/A	N/A
libgsm		5	0	4	1	8	8	0	0	0	0	0	0	0	0	N/A	N/A	N/A	N/A
libavc		5	0	1	4	5	0	5	1	0	0	0	0	0	0	N/A	N/A	N/A	N/A
libhevc		5	0	1	4	4	0	4	0	0	0	0	0	0	0	N/A	N/A	N/A	N/A
Total		3272	315	2639	318	3551	3269	282	440	413	27	4063	1164	2892	7	624	294	330	0

NEXZZER: #L: API misuse identified by *Length* rule; **#N**: API misuse identified by *Non-null* rule; **#R**: API misuse identified by *Range* rule; **#Mis-DU**: API misuse identified by "Missing Def-Use Edges" rule; **#I**: API misuse identified by "Invalid Edge" rule; **#FP**: suspicious vulnerabilities confirmed as API misuse after manual analysis; **#TP**: true positives, i.e., unique vulnerabilities after manual analysis;
Hopper: #Inf: automatically inferred API misuse cases;
UTopia: #FP: the number of unusable *drivers* containing API misuse code; **#TP**: correct *drivers* without API misuse;

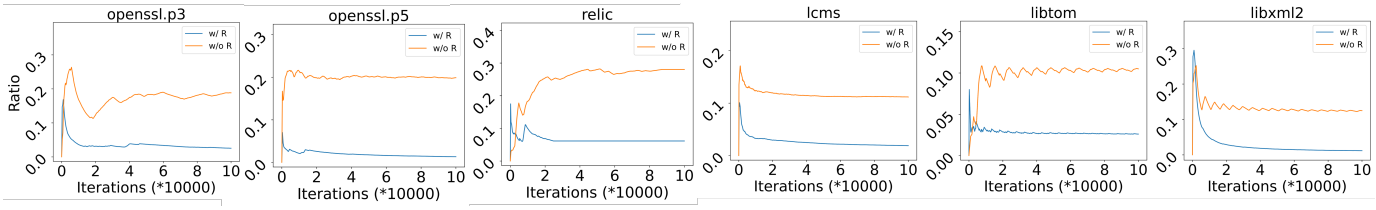

Fig. 3: API Misuse Ratio Trends (**w/o R**: NEXZZER* (without the relation learning to constrain the mutator); **w/ R**: NEXZZER)

Effectiveness of NEXZZER. We first discuss the overall effectiveness of NEXZZER to filter out API misuse. The results suggest that all rules (listed in **ValueSet** and **APIEdge** columns in Table IV) in §III-D4 can identify API misuse. However, some rules (e.g., Mis-DU) are more effective.

Comparison with Other Tools. We further compare NEXZZER with other tools regarding the number of crashes requiring human triage. Hopper identified a portion of (28.65% of) crashes as misuse. It still leaves 2,899 (71.35% of) crashes, of which only 7 were confirmed as vulnerabilities. This is because Hopper simply conducts type matching without considering certain implicit inter-API relations. For example, Hopper is not aware of the control dependencies for APIs such as *gmpz_init* in *libgmp* and *EVP_MAC_init* in *OpenSSL*, which should precede other APIs, resulting in extra API misuse crashes in these libraries.

We also compare NEXZZER with UTopia on the six libraries where UTopia is applicable. The methodology of UTopia to reduce API misuse is by exactly replicating unit test call sequences into multiple standalone drivers with only argument mutation. Therefore, in Table IV, we evaluate UTopia's true/false positives by manually analyzing how many **drivers** contain no API misuse and thus can effectively test the APIs. As shown by the results, a large portion of drivers (47.11%) produced by UTopia yield API misuse crashes because it incorrectly mutated certain API arguments. For example, UTopia falsely mutates an argument representing a file name or a *length* of buffers because it cannot identify the corresponding argument attributes in static analysis.

3) Implications of Misuse on Input Mutation: NEXZZER not only automatically identifies API misuse but also *gradually* generates fewer inputs that lead to API misuse. This is because APIGraph evolves with learned relations that constrain the mutation to avoid producing API misuse. To examine the effectiveness of relation learning in constraining API mutations, we measure the proportion of crashing inputs representing misuse among all fuzzing inputs (*i.e.*, **FP ratios**) and their changing trends. We compare NEXZZER with NEXZZER*, an implementation that uses *invariant* mutation strategies throughout the fuzzing process. Due to page limits, we list

partial results in Figure 3 and show all results in Figure 5 in the Appendix. In summary, the API misuse cases across different fuzzing iterations remain stable in NEXZZER*, yet there is a decrease of API misuse cases in NEXZZER as the fuzzing iterations progress. This demonstrates that NEXZZER is *dynamically* constraining mutations for producing valid inputs.

D. Unknown Misuse in NEXZZER and Practicability (RQ3)

In this subsection, we explain how we obtain the ground truth of whether a crash is misuse. We also discuss the unfiltered crashes leading to API misuse in NEXZZER and the human effort needed to validate them.

1) Methodology: To manually differentiate API misuse from vulnerabilities, we utilize a variety of comprehensive resources, such as documentation, library source code, consumer source code, public issues, and relevant discussions. Among these, we prioritize the official documentation and code comments. If these sources clearly state that specific constraints must be adhered to by API consumers, any crashes resulting from violations of such constraints are classified as API misuse, regardless of whether the constraints are contentious. This includes scenarios where the code invoking APIs should check for NULL pointer arguments prior to use.

We also encounter a lack of adequate official resources [45], [44]. To supplement our analysis, we examine existing consumer practices to determine if certain constraints are widely adopted. We also seek existing issues and discussions when encountering suspicious cases. For instance, in LCMS, NEXZZER identified a crash that initially appeared controversial; however, we discovered that the developer had previously addressed a similar issue in another API exhibiting analogous code patterns. These insights are useful for distinguishing true bugs that warrant fixing. Although this manual analysis can be labor-intensive, we have observed that, during fuzzing, most cases show recurring patterns (e.g., Mis-DU), and the number of contentious cases is relatively small.

In total, we spent approximately 90 hours analyzing all triggered crashes. Compared to Hopper, the effort required to analyze *one unknown crash* is similar, including debugging the API source code and examining the library documentation. The relation learning of NEXZZER further facilitates this process by providing hints to users about the crashing causes. We provide some examples in the Appendix §B. We believe our approach is more practical and scalable than generating monolithic drivers [15], [29] because NEXZZER can dynamically learn and filter misuse while monolithic drivers are not effective unless their misuse is manually fixed.

2) Reasons of API Misuse in NEXZZER: We investigated the causes of unrecognized API misuse in NEXZZER and concluded the following reasons.

Incomplete Rule-based Approach The first reason for API misuse in NEXZZER is that our rules in §III-D4 are incomplete. For example, in *tesseract*, NEXZZER triggered a crash. The crash was caused by the existence of an Invalid Edge from API *TessBaseAPIEnd* to *TessBaseAPIGetInputImage*, which NEXZZER successfully learned. However, the execution in *TessBaseAPIGetInputImage* does not trigger a Use-After-Free but a segmentation fault due to accessing null pointers cleared by *TessBaseAPIEnd*. This causes NEXZZER (and also Hopper [20]) to not categorize it as a UAF misuse.

Unrecognized Implicit Constraints Some more implicit API constraints are another source of unknown crashes in NEXZZER. For example, arguments in one API might have correlated constraints that NEXZZER cannot model, such as multiple length arguments for multi-dimensional arrays. Additionally, the method of type-matching (§III-C) might introduce over-approximated Def-Use edges. In OpenSSL, we might build a wrong edge from the return type of *BN_value_one* to the first parameter of *BN_add*. This is because *BN_value_one* returns a **constant** big number resides in the global static region, while *BN_add* tries to update it illegally.

Unsupported API Usage Some API usage is not supported by NEXZZER: APIs that trigger the *stdin* and are blocked during fuzzing, resulting in unrecognized timeout executions; mutating format strings of APIs such as *printf()* leads to crashes, which are also related to API misuse.

We believe that NEXZZER provides a foundation to support more complex constraints, including the cases mentioned above. This can be achieved by refining the types of APIGraph edges or adding support for edges between parameters within a single node, which we leave as future work.

VI. Related Work

API-Aware Fuzzing. Compared to standalone program fuzzing with one single entry to receive test inputs, API-aware fuzzing is used to test multiple interfaces together. Many researchers are focusing on API-aware fuzzing due to its inherent challenges of constraining various interdependent APIs. For example, for software libraries, the community has developed and maintained a large scale of effective fuzzing drivers [1], [25] for continuous testing. Furthermore, browser APIs [32], [30] and operating system kernels with system call interfaces are also popular targets [12], [13]. In the domain of web applications employing RESTful APIs [33], API-aware fuzzing is also widely applied. RESTler [31] can generate API requests based on the Swagger [4] specifications and guide mutations using service response feedback.

Fuzzing Automation. Fuzzing has been a hot research topic. FUDGE [16] uses a single library consumer to slice the Abstract Syntax Tree, extract multiple API call snippets, and create several small drivers. FuzzGen [15] constructs and merges API call graphs with control flows and data dependencies. Cross-platform initiatives like APICRAFT [19] and WINNIE [26] analyze execution traces on MacOS and Windows to generate fuzzing drivers. RUBICK [22] creates driver code for JAVA libraries using control-flow-sensitive analysis. DAISY [21] enhances driver sequence effectiveness by analyzing object resources. UTOPIA [29] uses unit test frameworks to reproduce API sequences and infer intra-API argument attributes. HOPPER [20] focuses on intra-API attributes with dynamic learning strategies. GraphFuzz [18] builds data-flow graphs for sequence mutation, needing

manually written schemas for accurate API dependencies. For OS targets, MoonShine [34] uses system call execution traces to produce high-quality input seeds. SyzGen [35] and KSG [36] assist kernel fuzzing by automatically generating system call descriptions from kernel source code.

VII. Conclusion

In this paper, we propose NEXZZER, an automatic library fuzzing framework that incorporates a hybrid strategy of static consumer analysis and dynamic relation learning to facilitate fuzzing automation on a wide range of library targets. NEXZZER also automatically identifies and filters out most false positive crashes caused by API misuse using an effective rule-based approach. The evaluation of NEXZZER demonstrates its significant improvements in the library code coverage and vulnerability-finding ability. Moreover, with NEXZZER, we detected 27 new vulnerabilities in widely used libraries including *OpenSSL* and *libpcre2*. 24 vulnerabilities were confirmed by the library developers and 9 were fixed because of our reports.

VIII. Acknowledgement

We thank the anonymous reviewers for their constructive comments on improving our work. This work is supported by the NSFC for Young Scientists of China (No.62202400) and the RGC for Early Career Scheme (No.27210024). Any opinions, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect the views of NSFC and RGC.

REFERENCES

[1] Google OSS-Fuzz: Continuous Fuzzing for Open Source Software. (https://github.com/google/oss-fuzz,2022), Accessed: 2023-Jan

[2] CVE-2024-6387. (https://nvd.nist.gov/vuln/detail/CVE-2024-6387,2024), Accessed: 2024-Jul

[3] Boost.org units module. (https://github.com/boostorg/units,2024), Accessed: 2024-Jul

[4] Swagger API Development for Everyone. (2023), https://swagger.io/

[5] Serebryany, K. Continuous fuzzing with libfuzzer and addresssanitizer. *2016 IEEE Cybersecurity Development (SecDev)*. pp. 157-157 (2016)

[6] Google Testing and Mocking Framework. Available at https://github.com/google/googletest

[7] Zalewski, M. AFL: American Fuzzy Lop. (https://lcamtuf.coredump.cx/afl/,2015), Accessed: 2023-Feb

[8] Google AddressSanitizer. Available at https://github.com/google/sanitizers/wiki/AddressSanitizer

[9] Rust Foreign Function Interface. Available at https://doc.rust-lang.org/nomicon/ffi.html

[10] Google fuzzer-test-suit. Available at https://github.com/google/fuzzer-test-suite/

[11] Google Syzlang. Available at https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md

[12] Choi, J., Kim, K., Lee, D. & Cha, S. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. *2021 IEEE Symposium On Security And Privacy (SP)*. pp. 677-693 (2021)

[13] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C. & Vigna, G. Difuze: Interface aware fuzzing for kernel drivers. *Proceedings Of The 2017 ACM SIGSAC Conference On Computer And Communications Security*. pp. 2123-2138 (2017)

[14] Google Syzkaller: Kernel Fuzzer. (https://github.com/google/syzkaller,2015), Accessed: 2023-Feb

[15] Ispoglou, K., Austin, D., Mohan, V. & Payer, M. Fuzzgen: Automatic fuzzer generation. *Proceedings Of The 29th USENIX Conference On Security Symposium*. pp. 2271-2287 (2020)

[16] Babić, D., Bucur, S., Chen, Y., Ivančić, F., King, T., Kusano, M., Lemieux, C., Szekeres, L. & Wang, W. Fudge: fuzz driver generation at scale. *Proceedings Of The 2019 27th ACM Joint Meeting On European Software Engineering Conference And Symposium On The Foundations Of Software Engineering*. pp. 975-985 (2019)

[17] Zhang, M., Liu, J., Ma, F., Zhang, H. & Jiang, Y. IntelliGen: Automatic Driver Synthesis for Fuzz Testing. *2021 IEEE/ACM 43rd International Conference On Software Engineering: Software Engineering In Practice (ICSE-SEIP)*. pp. 318-327 (2021)

[18] Green, H. & Avgerinos, T. GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs. *Proceedings Of The 44th International Conference On Software Engineering*. pp. 1070-1081 (2022)

[19] Zhang, C., Lin, X., Li, Y., Xue, Y., Xie, J., Chen, H., Ying, X., Wang, J. & Liu, Y. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries.. *USENIX Security Symposium*. pp. 2811-2828 (2021)

[20] Chen, P., Xie, Y., Lyu, Y., Wang, Y. & Chen, H. HOPPER: Interpretative Fuzzing for Libraries. *ArXiv*. abs/2309.03496 (2023), https://api.semanticscholar.org/CorpusID:261582917

[21] Zhang, M., Zhou, C., Liu, J., Wang, M., Liang, J., Zhu, J. & Jiang, Y. Daisy: Effective Fuzz Driver Synthesis with Object Usage Sequence Analysis. *2023 IEEE/ACM 45th International Conference On Software Engineering: Software Engineering In Practice (ICSE-SEIP)*. pp. 87-98 (2023)

[22] Zhang, C., Li, Y., Zhou, H., Zhang, X., Zheng, Y., Zhan, X., Xie, X., Luo, X., Li, X., Liu, Y. & Habib, S. Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation. *Proceedings Of The 32nd USENIX Conference On Security Symposium*. (2023)

[23] Fioraldi, A., Maier, D., Zhang, D. & Balzarotti, D. LibAFL: A Framework to Build Modular and Reusable Fuzzers. *Proceedings Of The 2022 ACM SIGSAC Conference On Computer And Communications Security*. pp. 1051-1065 (2022)

[24] Moroz, M. Structure-Aware Fuzzing with libFuzzer. (https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md,2015), Accessed: 2023-Feb

[25] Vranken, G. Differential fuzzing of cryptographic libraries. (https://github.com/guidovranken/cryptofuzz,2019), Accessed: 2023-Feb

[26] Jung, J., Tong, S., Hu, H., Lim, J., Jin, Y. & Kim, T. WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. (2021)

[27] Jiang, J., Xu, H. & Zhou, Y. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. *2021 36th IEEE/ACM International Conference On Automated Software Engineering (ASE)*. (2021)

[28] Takashima, Y., Martins, R., Jia, L. & Păsăreanu, C. SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis. *Proceedings Of The 42nd ACM SIGPLAN International Conference On Programming Language Design And Implementation*. (2021)

[29] Jeong, B., Jang, J., Yi, H., Moon, J., Kim, J., Jeon, I., Kim, T., Shim, W. & Hwang, Y. UTOPIA: Automatic Generation of Fuzz Driver using Unit Tests. *2023 2023 IEEE Symposium On Security And Privacy (SP) (SP)*. (2023)

[30] Zhou, C., Zhang, Q., Wang, M., Guo, L., Liang, J., Liu, Z., Payer, M. & Jiang, Y. Minerva: Browser API Fuzzing with Dynamic Mod-Ref Analysis. *Proceedings Of The 30th ACM Joint European Software Engineering Conference And Symposium On The Foundations Of Software Engineering*. (2022)

[31] Atlidakis, V., Godefroid, P. & Polishchuk, M. Restler: Stateful rest api fuzzing. *2019 IEEE/ACM 41st International Conference On Software Engineering (ICSE)*. pp. 748-758 (2019)

[32] Hodován, R. & Kiss, Á. Fuzzing javascript engine apis. *Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings 12*. pp. 425-438 (2016)

[33] Richardson, L. & Ruby, S. RESTful web services. (" O'Reilly Media, Inc.",2008)

[34] Pailoor, S., Aday, A. & Jana, S. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. *27th USENIX Security Symposium (USENIX Security 18)*. pp. 729-743 (2018,8)

[35] Chen, W., Wang, Y., Zhang, Z. & Qian, Z. SyzGen: Automated Generation of Syscall Specification of Closed-Source MacOS Drivers. *Proceedings Of The 2021 ACM SIGSAC Conference On Computer And Communications Security*. pp. 749-763 (2021), https://doi.org/10.1145/3460120.3484564

[36] Sun, H., Shen, Y., Liu, J., Xu, Y. & Jiang, Y. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. pp. 351-366 (2022,7), https://www.usenix.org/conference/atc22/presentation/sun

[37] Blondel, V., Guillaume, J., Lambiotte, R. & Lefebvre, E. Fast unfolding of communities in large networks. (J. Stat. Mech. (2008) P10008,2008)

[38] Sun, H., Shen, Y., Wang, C., Liu, J., Jiang, Y., Chen, T. & Cui, A. HEALER: Relation Learning Guided Kernel Fuzzing. *Proceedings Of The ACM SIGOPS 28th Symposium On Operating Systems Principles*. pp. 344-358 (2021), https://doi.org/10.1145/3477132.3483547

[39] Project, L. The LLVM Compiler Infrastructure. (2023), Available at https://llvm.org/

[40] Project, L. SanitizerCoverage, Accessed: 2023-Dec, https://clang.llvm.org/docs/SanitizerCoverage.html

[41] Lattner, C. & Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings Of The 2004 International Symposium On Code Generation And Optimization (CGO'04)*. (2004,3)

[42] Litvak, S., Dor, N., Bodik, R., Rinetzky, N. & Sagiv, M. Field-sensitive program dependence analysis. *Proceedings Of The Eighteenth ACM SIGSOFT International Symposium On Foundations Of Software Engineering*. pp. 287-296 (2010)

[43] Andersen, L. & Lee, P. Program Analysis and Specialization for the C Programming Language. (2005), https://api.semanticscholar.org/CorpusID:20876553

[44] Schlichtig, M., Sassalla, S., Narasimhan, K. & Bodden, E. FUM - A Framework for API Usage constraint and Misuse Classification. *2022 IEEE International Conference On Software Analysis, Evolution And Reengineering (SANER)*. pp. 673-684 (2022)

[45] Gu, Z., Wu, J., Liu, J., Zhou, M. & Gu, M. An Empirical Study on API-Misuse Bugs in Open-Source C Programs. *2019 IEEE 43rd Annual Computer Software And Applications Conference (COMPSAC)*. **1** pp. 11-20 (2019)

[46] Pacault, J. Computing the Weak Components of a Directed Graph. *SIAM Journal On Computing*. **3**, 56-61 (1974), https://doi.org/10.1137/0203005

APPENDIX

A. Vulnerability-Finding Abilities

We provide the deduplicated vulnerabilities newly detected by NEXZZER in Table V.

We also provide the results of different library fuzzers in reproducing known vulnerabilities in Figure 4 and Table VI. We use the Google Fuzzer Test Suite (FTS) [10] as the benchmark of vulnerability reproduction. As the result shows, NEXZZER excels at discovering known library vulnerabilities.

Besides the concluded reasons of better accuracy and diversity (§V-B), better automation is also a crucial reason for NEXZZER to reveal previously unknown vulnerabilities. More than half of the vulnerabilities in Table V lies in APIs that are poorly or even not covered in existing fuzzing drivers or unit tests, which demonstrates NEXZZER's advantages of automation and scalability. NEXZZER also find deep vulnerabilities in covered APIs. We provide details of two interesting cases below, which have been fixed in the latest version of libraries.

```
1  r0 = BN_new()
2  r1 = BN_MONT_CTX_new()
3  r2 = BN_CTX_new()
4  BN_sub_word(r0, 0x1)
5  BN_set_bit(r0, 0x1e4abbe6) // set a large integer
6  r3 = BN_dup(r0)
7  r4 = BN_dup(r0)
8  r5 = BN_dup(r0)
9  BN_MONT_CTX_set(r1, r5,r2)
10 BN_mod_exp_mont_consttime(r0, r3, r4, r5,
11     r2, r1)              // trigger an integer overflow
```

Listing 8: PoC seed 1

```
1  r0=pcre2_general_context_create(0x0, 0x0, "}@N\x00")
2  r1=pcre2_compile_context_create(r0)
3  r2=pcre2_compile("}@N\x00", 0x4, 0x4000000,
4      &(0x2000100)=0x4000000, &(0x2000140)=0x0, r1)
5  r3=pcre2_match_data_create(0x1000000,
6      r0, r2)         // trigger an invalid type conversion
7  r4=pcre2_match_context_create(r0)
8  pcre2_match(r2, "}@N\x00", 0x4, 0x0,
9      0xe, r3, r4)    // heap buffer overflow
```

Listing 9: PoC seed 2

An integer overflow in OpenSSL. The first seed shown in Listing 8 is an integer overflow from the OpenSSL's Big Number APIs. The API *BN_set_bit* in line 5 sets a member in the *r0* to a very large integer. Then inside *BN_mod_exp_mont_const* (line 10), the API would calculate a buffer size based on the member of *r0* with a limit check. However, if the required size is overflowed to a negative integer, thereby bypassing the limit check, it leads to potential buffer overflow or Denial-Of-Service.

The vulnerability is difficult for existing approaches to explore because manual or synthesized drivers usually cover a highly limited input space. For example, in manual drivers related to the Big Number APIs in Listing 7, they set up random BIGNUM content with a byte array by using the *BN_bin2bn* API, making it impossible to overflow the vulnerable variable since fuzzing engines cannot produce such large byte arrays. More importantly, large-scale APIs make it extremely challenging for consumers to cover enough sequence usage.

TABLE V: Newly Discovered Vulnerabilities

Targets	Type	Function	Status	ID
OpenSSL	Integer Overflow	BN_mod_exp_mont_consttime	Fixed	4378e
OpenSSL	Integer Overflow	BN_bntest_rand	Fixed	23704
OpenSSL	Integer Overflow	ASN1_BIT_STRING_set_bit	Fixed	20719
OpenSSL	Integer Overflow	ASN1_BIT_STRING_get_bit	Fixed	20719
OpenSSL	Undefined Behavior	BN_GF2m_mod_inv	Reported	19826
OpenSSL	Integer Overflow	RC2_ofb64_encrypt	Confirmed	22986
OpenSSL	Stack Overflow	RC2_cfb64_encrypt	Confirmed	22986
OpenSSL	Integer Overflow	TXT_DB_create_index	Confirmed	22986
OpenSSL	Integer Overflow	TXT_DB_get_by_index	Confirmed	22986
OpenSSL	Stack Overflow	CAST_ofb64_encrypt	Confirmed	22986
OpenSSL	Integer Overflow	CAST_cfb64_encrypt	Confirmed	22986
OpenSSL	Stack Overflow	DES_ede3_ofb64_encrypt	Confirmed	22986
OpenSSL	Stack Overflow	DES_ofb64_encrypt	Confirmed	22986
OpenSSL	Stack Overflow	DES_cfb64_encrypt	Confirmed	22986
OpenSSL	Stack Overflow	DES_ede3_cfb64_encrypt	Confirmed	22986
OpenSSL	Stack Overflow	RC5_32_ofb64_encrypt	Confirmed	22986
OpenSSL	Stack Overflow	RSA_padding_add_PKCS1_type_1	Confirmed	22986
OpenSSL	Stack Overflow	BF_ofb64_encrypt	Confirmed	22986
libxml2	Null Pointer Dereference	xmlCopyNode	Confirmed	463
Relic	Integer Overflow	bn_grow	Fixed	CVE-2023-36326
Relic	Heap Overflow	bn_get_prime	Fixed	CVE-2023-36327
libTom	Integer Overflow	mp_grow	Fixed	CVE-2023-36328
libpcre2	Heap Overflow	pcre2_match_data_create	Fixed	CVE-2023-29822
libGMP	Integer Overflow	mpz_nextprime	Fixed	CVE-2022-46386
lcms	Buffer Overflow	cmsCreateExtendedTransform	Fixed	46355
libopus	Stack Overflow	opus_decode	Reported	329
tesseract	Integer Overflow	TessBaseAPIAdaptToWordStr	Reported	4299

There are hundreds of Big Number APIs, and the vulnerable sequence in Listing 8 does not appear in any consumer.

An heap overflow in PCRE2. Listing 9 shows a heap overflow in PCRE2. In line 5, the first argument of *pcre2_match_data_create* is passed in as a 32-bit integer. However, it is implicitly cast to a 16-bit integer in a member of *r3*, which results in an integer overflow but would not crash instantly. Subsequently, in *pcre2_match* (line 8), the overflowed member causes a heap buffer overflow vulnerability. The vulnerable arguments in Listing 9 are usually treated as constants and remain untested in existing fuzzing drivers.

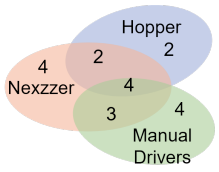

Fig. 4: The Venn diagram of found known vulnerabilities in FTS.

	Manual	Hopper	NEXZZER
lcms	1	4	4
libxml2	4	1	2
libpcre2	2	2	3
openssl	2	0	2
boringssl	1	0	1
harfbuzz	1	1	1
Total	11	8	13

TABLE VI: Numbers of found known vulnerabilities in different libraries in FTS.

B. API Relation Case Studies

```
1  BN_mod_exp_mont_consttime::SeedSpace[(Listing 8)]:
2    Value_Set: BN_set_bit::arg1::[0, 0x80000001]
3    Invalid_Edges: BN_set_bit -> BN_mod_exp_mont_consttime
4
5  pcre2_match::SeedSpace[(Listing 9)]:
6    Value_Set: pcre2_match_data_create::arg0::[0, 0xffff]
7
8  APIGraph::Invalid_Edges:
9    BN_free::arg0       -> BN_add::arg0
10   BN_free::arg0       -> BN_sub::arg0
11   BN_generate_prime   -> BN_add_word    (suspicious bug)
12   ...
13 APIGraph::Valid_Edges:
14   EVP_DecryptInit::arg0      -> EVP_DecriptUpdate::arg0
15   RSA_generate_key_ex::arg0 -> RSA_private_decrypt::arg3
16   xmlUnlinkNode      -> xmlAddChild      (suspicious bug)
17   xmlUnlinkNode      -> xmlAddChildList (suspicious bug)
18   ...
```

Listing 10: Seedspaces and Relations Example

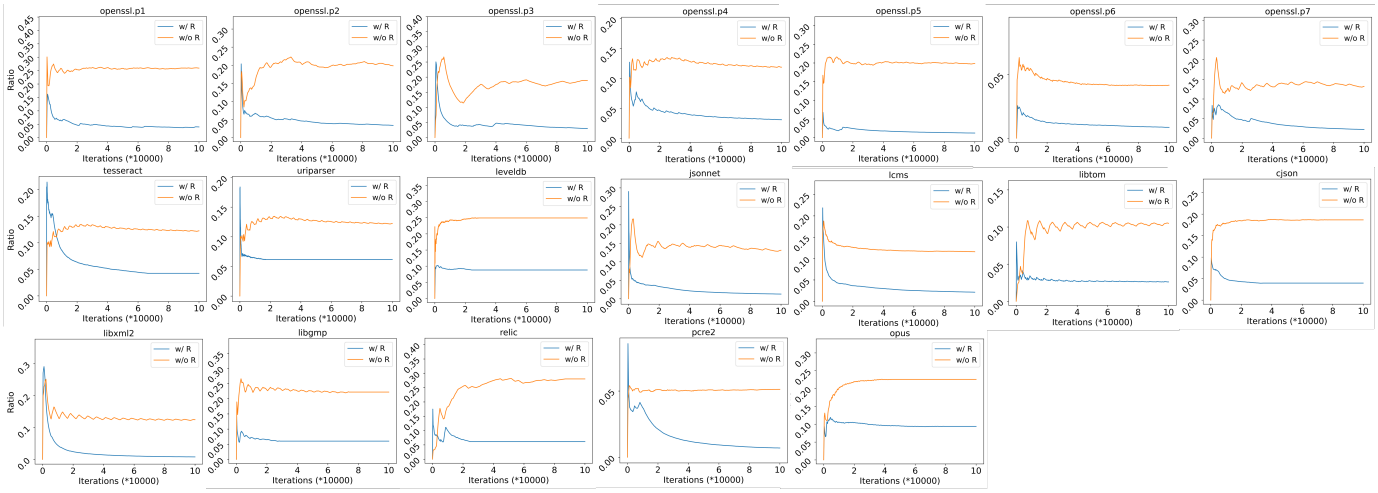

Fig. 5: API misuse (i.e., crashing inputs leading to API misuse) Ratio Trends

Listing 10 shows examples of API relations learned by NEXZZER. Line 1 to 6 shows the SeedSpaces stored in the API nodes representing the two bugs described above (subsection A). Based on the *Value-Set* constraints (i.e., the seed is crashing outside the value range), one can easily infer that there are likely integer overflow vulnerabilities during manual analysis.

Lines 9 to 11 show several learned Invalid Edges. The first two edges represent identified API misuse (Use-After-Free) during fuzzing, while the third edge (from *BN_generate_prime* to *BN_add_word*) is counterintuitive and suspicious. After manual analysis, we discovered that *BN_generate_prime* would implicitly *free* the *BIGNUM* structure passed in if some internal calls failed and thus cause a UAF in *BN_add_word*. Such behavior is not well-designed and has been deprecated. Lines 14 to 15 are examples of several Valid Edges that represent correct API usage, where the source API is necessary for the target API's intended usage. However, lines 16 and 17 imply a controversial design in libxml2's API *xmlAddChild* and *xmlAddChildList*. It indicates that if an XML node is about to be added to a tree, it should be a new or unlinked node instead of a linked node in a tree. While the developer confirmed that the ideal implementation should handle such cases, it is assessed and will not be fixed due to historical internal compatibility.

These cases illustrate that NEXZZER's relation learning not only facilitates vulnerability analysis but also provides instructive knowledge to understand and refine the API designs.

Modeling API relations by the APIGraph edges facilitates effective API sequence generation (§V-B3) and is also used to design certain filtering rules for common misuse. As described in §III-D4, we filter edge-related misuse including the miss of Def-Use edges and the presence of certain Invalid edges (*i.e.*, UAF misuse). However, we do not attempt to identify whether a Valid Edge indicates API misuse or a potential vulnerability currently. Because, based on our practice, Valid Edges are hard to categorize by heuristic rules and their number is relatively small, which makes it feasible for manual triage. Therefore, more effective methods of filtering misuse based on Valid Edges are a potential improvement for NEXZZER, such as improving the static analysis on consumers or utilizing other extra resources like a language model.

Algorithm 1: Node Insertion

Input: APIGraph G, Seed S, Position P
Output: New Seed S'
$invalid\_nodes \leftarrow \emptyset$;
$valid\_nodes \leftarrow \emptyset$;
for $prev\_node \in S.nodes[0..P]$ **do**
  for $edge \in G[prev\_node].out\_edges$ **do**
    if $edge.type = INVALID$ **then**
      $invalid\_nodes \leftarrow invalid\_nodes \cup edge.node$;
  end
end
for $post\_node \in S.nodes[P..end]$ **do**
  for $edge \in G[post\_node].in\_edges$ **do**
    if $edge.type = INVALID$ **then**
      $invalid\_nodes \leftarrow invalid\_nodes \cup edge.node$;
  end
end
$insert\_node \leftarrow random\_choose(G.all\_nodes \setminus invalid\_nodes)$;
for $edge \in G[insert\_node].in\_edges$ **do**
  if $edge.type = VALID$ **then**
    $valid\_nodes \leftarrow valid\_nodes \cup edge.node$;
end
$S' \leftarrow S.insert(node, valid\_nodes)$;

C. NEXZZER *Algorithms*

We describe several key algorithms in NEXZZER's fuzzing process in this section, including API node insertion, seed

Algorithm 2: Learning relations across APIs

```
1  Input: APIGraph G, API Sequence S, target API
     index idx
2  status ← execute(S).status
3  E ← ∅
4  for i ∈ idx-1 → 0 do
5      for j ∈ i → idx-1 do
6          E_du ←
             S.find_edges(S.node_i, S.node_j, Defuse)
7          E_dep ←
             S.find_edges(S.node_i, S.node_j, Dep)
8          if E_du ∪ E_dep = ∅ then
9              S' ← S.remove(i)
10             status' ← execute(S').status
11             if status' ≠ status then
12                 E ← E ∪ Edge(S.node_i, S.node_idx)
13             else
14                 S ← S'
15             end
16     end
17 end
18 G.update(S, E)
```

Algorithm 3: A Fuzzing Iteration

```
1  Input: APIGraph G, Seed Corpus C.
2  ▷ E: learned edges.
3  ▷ X: learned argument constraints.
4  ▷ S: current seed.
5  S ← mutation_generation(APIGraph, C)
6  feedback ← execute(S)
7  idx ← feedback.target_idx
8  SS ← G.node_idx.SeedSpaces
9  if duplicate(S, idx, SS, feedback) then
10     return
11 S, E ← InterAPI_relation_learning(G, S, idx)
12 X ← IntraAPI_relation_learning(S, idx)
13 SS.update(S, X, E)
14 for e ∈ RULES do
15     check_misuse(e, S, SS)
16 end
17 G.update(SS)
```

minimization, and a complete fuzzing iteration.

1) API Node Insertion: When inserting an API call into a Liblang seed, NEXZZER's mutator guarantees that **no** inter-API misuse encoded in the APIGraph will be introduced into the mutated seed after insertion.

Specifically, algorithm 1 takes a randomly chosen position *P* in a seed *S* and the APIGraph *G* as inputs. The algorithm first finds out nodes that are invalid to be inserted at *P*. The invalid nodes include all API nodes in *G* that can be reached by any node in *S* that lies **before** *P* through invalid edges (line 5-10). The invalid nodes also include all API nodes in *G* that can reach to any node in *S* that lies **after** *P* through invalid edges (line 11-16) Excluding these invalid nodes when choosing a candidate node for insertion (line 17) guarantees that no invalid edge is introduced to the new seed. The algorithm also finds out known valid nodes that connected to the candidate node for insertion. Line 22 inserts the candidate node with its valid nodes together.

2) Learning Inter-API Relations: Algorithm 2 shows NEXZZER's algorithm of learning inter-API relations. Besides the description in §III-D3, it is worth noting that there are benefits of performing the removal *from back to forth* (line 7). During the removal, we do not remove nodes that have Def-Use out-edges to not only N_t but **any** subsequent node preserved in the seed. Because subsequent nodes that are already preserved indicate their direct/indirect relations to N_t. If a node N is preserved during the back-to-forth iteration, and a previous node N_p has direct Def-Use relation to N, this means N_p has indirect relations to N_t through N and should not be removed. For example, in *r1=f1(); r2=f2(r1); f3(r2)*, although *f1* has no direct relations to *f3* (N_t), it can not be removed because it has indirect relations through *r1* in *f2* to *f3*. The back-to-forth iteration facilitates the maintenance of such indirect relations.

3) A Fuzzing Iteration: Algorithm 3 depicts the high-level process of a fuzzing iteration in NEXZZER. It first randomly applies either the generation or mutation strategy (§III-D1 and §III-D2) to produce and execute a new seed (lines 5-6). Based on the execution status, it obtains the target API node's index in the seed (line 7), which either triggers new coverage or crashes. The deduplication (line 10) checks whether previous seeds stored in the SeedSpace (line 9) have similar crashes with the current seed (§III-D3) and subsequently learns inter/intra- API relations. The learned relations and feedback are saved in the SeedSpace (line 14). Finally, for each crash, utilizing its learned relations, NEXZZER identifies if it is misuse based on the predefined filtering rules (lines 15-17). The SeedSpace is updated back to the APIGraph with new relations (line 18).