UTOPIA: Automatic Generation of Fuzz Driver using Unit Tests

Bokdeuk Jeong[†] , Joonun Jang[†] , Hayoon Yi[†] , Jiin Moon[†] , Junsik Kim[†] , Intae Jeon[†] ,
Taesoo Kim[†⋆], WooChul Shim[†*], Yong Ho Hwang[†]
bd.jeong, joonun.jang, hayoon.yi, jiin.moon, junsik1.kim, intae.jeon, tsgates.kim, woochul.shim, yongh.hwang@samsung.com

[†] *Samsung Research, Republic of Korea*
[⋆] *Georgia Institute of Technology, USA*

***Abstract*—Fuzzing is arguably the most practical approach for detecting security bugs in software, but a non-trivial extent of efforts is required for its adoption. To be effective, high-quality fuzz drivers should be first formulated with a proper sequence of APIs that can exhaustively explore the program states. To alleviate this burden, existing solutions attempt to generate fuzz drivers either by inferring the valid sequences of APIs from the consumer code (i.e., actual uses of APIs) or by directly extracting them from sample executions. Unfortunately, all existing approaches suffer from a common problem: the observed API sequences, either statically inferred or dynamically monitored, are intermingled with custom application logics. However, we observed that the unit tests are carefully crafted by the actual designer of the APIs to validate their proper usages, and importantly, it is a common practice to write the unit tests during their development (e.g., over 70% of popular GitHub projects).**

In this paper, we propose, UTOPIA, an open-source tool and analysis algorithm that can automatically synthesize effective fuzz drivers from existing unit tests with near-zero human involvement. To demonstrate its effectiveness, we applied UTOPIA to 55 open-source project libraries, including Tizen and Node.js, and automatically generated 5K fuzz drivers from 8K eligible unit tests. In addition, we executed the generated fuzzers for approximately 5 million per-core hours and discovered 123 bugs. More importantly, 2.4K of the generated fuzz drivers were adopted to the continuous integration process of the Tizen project, indicating the quality of the synthesized fuzz driver. The proposed tool and results are publicly available and maintained for a broader adoption among both researchers and practitioners.

I. INTRODUCTION

Fuzzing is an effective technique for automatically discovering software bugs and security vulnerabilities. For instance, Google's OSS-Fuzz [2], that provides continuous fuzzing for open-source software, discovered more than 30,000 bugs in 500 projects since its announcement in 2016. The basic principle of fuzzing is simple: it generates and feeds pseudorandom inputs to a given program and checks if the program properly processes the provided inputs during execution. If the program behaves abnormally (e.g., crashes or hangs) during the execution, it provides the bug-triggering inputs as the proof of existence of a software bug.

Essentially, two types of fuzzers exist: an end-to-end fuzzer that aims to test an entire program as a blackbox (e.g., afl [27]), and a library fuzzer that focuses on testing specific interfaces or APIs of the library (e.g., libFuzzer [25]). As end-to-end fuzzing can enable automatic testing without requiring considerable efforts from the developer for adoption, it has become widely popular, especially when the fuzzer is integrated for the first time. In contrast, a library fuzzer requires manual and deep integration with the library, called *fuzz drivers*, which describe a sequence of API calls processing the fuzzer-provided inputs. Upon the maturity of a project, such a deep integration is preferred by the developers as its whitebox nature allows deeper state exploration.

Recently, researchers have been exploring a means of alleviating the burden of manual integration for library fuzzers by automatically generating or synthesizing fuzz drivers [12, 16, 18, 28]. They formulate a proper sequence of APIs for fuzzing either by stochastically inferring API dependencies from their uses in the source code [12, 16], or from the execution traces observed during runtime [18, 28]. More specifically, one of the original projects, Fudge [12], focuses on directly reducing a proper sequence of API calls from the *consumer* code, wherein the API uses and the custom application logics are intermingled. A following project, FuzzGen [16], can theoretically produce reasonably efficient fuzz drivers (i.e., inferring a valid sequence of API calls) by performing a *whole* program analysis that reasons about the API dependencies from the *multiple* consumer codes. Although these approaches are general purpose and broadly applicable, they rely on *consumer* code that exhibits fundamental limitations. In particular, the intermixed code may end up generating simple API sequences or semantically invalid states (e.g., allocation in one code and uses in other places). Inferring valid API sequences and dependencies from statistical aggregation of consumer code would render stereotypical cases that are *not* ideal for fuzzers looking for invalid, uncommon inputs.

Unlike existing projects that attempt to infer the API dependencies, we utilized the exact sequence of API calls in *unit tests (UTs)*. We observed that 1) the existing UTs explicitly convey such dependencies on APIs that a developer cares about, 2) UTs check various facets of the functionality provided by a library with more APIs (e.g., internal ones) than *consumer* code, and 3) numerous existing projects already have well-written UTs—73% from Github on average, Android external and OSS-Fuzz as listed in Table I. In addition, we

*Corresponding Author.

Category	Total \#	Unit Tested \# (rate)	Fuzz Tested \# (rate)
GitHub C/C++ (Top 200)	200	143 (72%)	38 (19%)
Android External	316	224 (71%)	61 (19%)
OSSFuzz	450	347 (77%)	450 (100%)

Table I: Over 70% of popular open-source projects in GitHub contain UTs, but approximately 20% of these projects implemented fuzz drivers. We aim to bridge this gap by automatically converting existing UTs to fuzz drivers.

observed that a non-trivial number of open-source projects—nearly 80% of popular open-source projects in GitHub (Table I) have not adopted fuzzing as a means for testing to date.

In this paper, we propose UTopia that employs techniques to convert each existing UT to an effective fuzz driver in an automated and scalable manner. The key ideas behind UTopia are to 1) leverage UT specific properties to unravel the complexity in UT analysis, 2) perform *root-definition analysis*, a newly introduced technique, to trace back the source of API arguments for proper fuzz input injection that maintains the inter-procedural relations and data-flow intended by the developers, and 3) reflect in fuzz input mutation, the analysis of impacts each argument may have within the internals of its API. This enables UTopia to deeply explore the code space and avoid crashes resulting from invalid API usage. Therefore, a push-button solution can be provided to automatically synthesize high-quality fuzz drivers with no human involvement.

To demonstrate the effectiveness of automatic fuzz driver synthesis, we applied UTopia to 55 open-source project libraries (25 projects from Table I and 30 from Tizen) among projects that satisfy the following conditions: 1) written in C/C++, 2) uses gtest [7], boost [6], or TCT [5] for UT, and 3) supports LLVM build on Linux. We selected 25 projects, considering the diversity of their project size, building systems and UT frameworks, to evaluate UTopia by testing with their generated fuzz drivers. For the 30 Tizen projects, the generated fuzz drivers have been adopted by the Tizen community and are continuously executed [4], verifying the quality of the generated fuzz drivers.

With the use of UTopia, we could find and report a total of 123 bugs in popular open-source projects such as Tizen and libaom. Among these bugs, 70 were confirmed by the project communities, 48 are reported but await community response, and the remaining 5 were in exposed APIs that were considered internal by the developers and determined as not required to be fixed.

In this paper, we make the following contributions:

- We propose a new fuzz driver synthesis approach that embraces the existing unit tests for automatically generating fuzz drivers.
- We implemented a prototype of the present approach, UTopia, for C/C++ libraries utilizing the `gtest` and `boost` UT frameworks and empirically validate that it can successfully transform 5K UTs into meaningful fuzz drivers for 55 open-source project libraries. UTopia is available publicly at *https://github.com/Samsung/UTopia*.
- We have reported 123 new bugs, among which 70 were confirmed during the responsible disclosure process (Table VII).

II. Challenges and Proposed Approach

UTopia aims to generate high-quality fuzz drivers in a completely automated and scalable manner. To achieve this goal, UTopia should minimize the human involvement during the entire generation process by addressing two major challenges:

C1: Synthesize valid API call sequences
C2: Synthesize valid API call parameters

This is due to the fact that, in a fuzz driver, libraries can crash not only because a provided input hits a bug but also because of invalid API usage caused by C1/2. Such unintended crashes, referred to as *spurious* crashes, make the fuzzing process ineffective; it diverts the fuzzer to an uninteresting state and demands manual analysis of found crashes. In severe cases, they can even call for manual fixing of drivers, because they might prevent a driver from performing any meaningful fuzzing exploration (e.g., a driver calling a dereferencing API on an uninitialized pointer will assuredly crash with little chance of exploration).

In §II-A and §II-B, we describe C1 and C2 with additional difficulties related to these challenges and introduce UT as the key enabler to UTopia sidestepping such problems. In §II-C, we address that the seemingly simple idea of using UT presents additional interesting challenges.

Running example. We describe the challenges in this section by using a running example in Figure 1, a UT containing the API uses for reading or writing raw data in the OpenCV library, with which UTopia could generate a fuzz driver reproducing CVE-2019-5063. Basically, UTopia transforms the UT into a fuzz driver by making minor changes to the original UT code in order to assign fuzz input to the existing variables that are analyzed as sources of API arguments. For instance, in Figure 1, UTopia respectively transforms lines 5, 9, 14, 22 into lines 6, 10, 18, 23 and inserts an assignment statement (line 17) to provide fuzz input to variables that influence the call parameters of APIs. Note that certain parameters of the APIs are deliberately not fuzzed, because UTopia analyzes that changing them could potentially cause severe spurious crashes (details of exclusion criteria are elaborated in §III-D).

A. Synthesizing Valid API Call Sequences

A major challenge in fuzz driver generation is to figure out which library APIs to call and in what order to call them because APIs may often have strict order dependencies as observed in our running example: `FileStorage()` → `writeRaw()` → `release()`. Therefore, the construction of random sequences of APIs for fuzz drivers might simply waste the fuzzing effort (e.g., any crash resulting from `writeRaw()` calls after a `release()` without a constructor call would be considered a spurious crash rather than a bug.)

```
 // ref. @OpenCV:modules/core/test/test_io.cpp
 TEST(Core_InputOutput, filestorage_base64_basic_rw_XML) {
   std::vector<data_t> rawdata;

-  std::string name = "test.xml";
+  std::string name = fi1;

+  // NB. a loop parameter is bounded to avoid timeout
-  const size_t rawdata_N = 40;
+  const size_t rawdata_N = fi2;

   /* a 4d mat */
   const int dim[] = {4, 4, 4, 4};
-  Mat m(4, dim, CV_64FC4, cvScalar(0.8, 0.1, 0.6, 0.4));

+  // NB. the len of dim is correctly inferred and respected
+  for (unsigned i = 0; i < fi3_size; ++i) { dim[i] = fi3[i]; }
+  Mat m(fi3_size, dim, CV_64FC4, cvScalar(fi4, fi5, fi6, fi7));

   /* raw data */
   for (int i = 0; i < (int)rawdata_N; i++) {
-    data_t tmp; tmp.u = 1; tmp.d = 0.1;
+    data_t tmp; tmp.u = fi8; tmp.d = fi9;
     tmp.i = i; rawdata.push_back(tmp);
   }

   /* write */ // NB. not fuzzing: file mode
   FileStorage fs(name, FileStorage::WRITE_BASE64);
   fs << m;
   for (int i = 0; i < (int)rawdata_N; i++)
     fs.writeRaw(data_t::signature(), &rawdata[i],
                 sizeof(data_t));
   fs.release();

   /* read */ // NB. not fuzzing: file mode
   FileStorage fs(name, FileStorage::READ);
   fs.readRaw(data_t::signature(), &rawdata[0],
              rawdata.size() * sizeof(data_t));
   fs.release();
 }
```

Figure 1: Running example. Simplified UT from OpenCV tests `FileStorage` by storing and reloading matrix data encoded as XML. Based on this UT, UTopia generated a fuzz driver (differences marked with -/+) that can trigger an XML parsing error, previously reported as CVE-2019-5063 [1]. The global variables, `fi{1-9}`, contain the mutated fuzzing inputs for each run.

Prior research have opted for extracting [12] or inferring [16] valid API sequences from the general consumer code. However, this decision introduces its own difficulties (**D**).

D1: Decisions for consumer code analysis. If any given consumer code is to be accommodated, as done in prior studies, the analysis procedure of the consumers should be decided. A potential approach could involve an analysis across the entire consumer code, which enables the extraction of entire API usage patterns within the consumer. However, in case of encountering complex consumer code with numerous API calls scattered across complicated control flows, the extracted patterns can become bloated. This causes a driver to call in dozens or hundreds of API calls, which hinders the fuzzing efforts of the driver owing to the bloated input space from the multitude of APIs.

To avoid such a case, another approach involves limiting the amount of consumer code considered for generating a single fuzz driver (e.g., prior work [12, 16] implemented their work to analyze only across the same compilation unit to extract the usage of library API calls). Although this mitigates the earlier problem of bloated sequences, this may cause the acquirement of incomplete API sequences due to the necessary API calls residing in various source files. Moreover, generating a fuzz driver from such a sequence may yield spurious crashes.

Proposed approach. Unlike a prior research that focuses on correctly reconstructing valid API sequences from any given consumer, we take advantage of explicit API sequences written in unit tests to completely sidestep the aforementioned challenges of API sequence synthesis.

The advantages of using UT are twofold:

1) Explicit construction of the library state per test case in UT means no burden of API pattern inference or extraction for generating fuzz drivers
2) This is consistent with the purpose of the fuzz driver in which each test case, and its contained API sequence, is designed to test a specific property/invariant of the library deemed essential by library developers

In our running example, this approach enables us to maintain all API order relations within the code as we do not alter any of the calls. Additionally, as the UT only contains the required short API sequences to test specific properties of a library, UTopia is not prone toward generating bloated API sequences.

B. Synthesizing Valid API Call Parameters

Another challenge in fuzz driver generation is to understand the intra- and inter-API logic and appropriately assign fuzz input values according to their semantic relationships. Otherwise, providing random fuzz values for all parameters of API calls would cause issues with spurious crashes and wasted fuzz input, which eventually degrades the fuzzing capability of the driver. Therefore, the following difficulties must be addressed to provide suitable fuzzing values for APIs and minimize spurious crashes.

D2: Inferring inter-API semantics. Objects or values are often shared via parameters between library APIs and contain the information required by the APIs to perform their functions in a consistent context. Such relations across the API parameters should be maintained in generating fuzz drivers to ensure valid API usage. In our running example, the `FileStorage` object `fs` would be such the case, which the library class APIs of `FileStorage` are all called on.

The primary relations between the APIs requiring valid semantics are:

- *out-to-in:* an output of an API is used as an input argument in another API;
- *fixed:* arguments in each APIs should be identical across the API calls;
- *relative:* arguments in different APIs are derived from the same value (e.g., `x=f(y); API_1(x); z=x+g(y); API_2(z);`)

The inter-API relations cannot be accurately analyzed from its use in general consumer code using a conventional intra-procedural data flow analysis, because if we enter a fuzz input without considering inter-procedural flow, the developer's intended data flow may be neglected (e.g., in `var a=3; → b=func(a); → Target_API(b);`. Conversely, relying only on intra-procedural analysis will impose the assignment of the fuzz input to `b` instead of `a`). Alternatively, the relation can be inferred via argument type aliases between the APIs [16], but it cannot be assured whether they refer to the same object. In these cases, the resulting fuzz driver cannot reflect the inter-API relationships between the parameters.

D3: Inferring intra-API semantics. In addition, the relationships between the arguments within the same API should be considered. The most common considerations include those regarding arrays:

- *array↔length:* an input parameter indicates the length of another input array parameter;
- *array↔index:* an input parameter is an index of another array input parameter;

For instance, the first argument in the `Mat` class constructor (line 14 in Figure 1) demands correspondence between the size of the arrays stated in the second and fourth argument. If these were randomly fuzzed, the driver would mostly either result in a segfault (size argument > actual size of array) or wasting effort on mutating unused fuzz input bytes (size argument < actual size of array). If the type-based pattern matching approach of Fudge [12] is applied on this example, it would be unable to match the logical relation between the first and fourth parameters, because such a relation is not explicitly exposed in the consumer code. Although, the value-set approach of FuzzGen [16] can perform analyses on the internals of the APIs, they are prone to failure in representing the relationship between the three arguments because it infers the type and value-set of the individual arguments rather than the relationship between the arguments.

D4: Detrimental input for fuzzing. Although not considered API misuse, we have observed that there are some arguments that, when fuzzed without care, degrade fuzzing performance. For instance, if an argument is used for memory allocation or loop count, large values frequently result in out-of-memory or timeout errors respectively. Although these are not spurious crashes, they often similarly hinder further fuzzing exploration.

Proposed approach. To maintain valid argument semantics, UTopia preserves the original data flow in UT and finds where to inject fuzz input (i.e., fuzz target) and how they should be mutated (API attribute) through static analysis. To recognize the suitable locations to inject the fuzz input, we defined a new concept of root definition that is an assignment statement in which a variable is defined with a constant. By assigning fuzz input at only root definitions, we can preserve the original data flow and naturally adhere to the existing inter-API semantics because the flow between the parameters of APIs is uninterrupted. In Figure 1, UTopia delivers fuzz input to the third argument `rawdata` in the writeRaw() API (line 31) by assigning fuzz input to the root definition (line 23) where each element of the vector `rawdata` is assigned with constant.

Upon locating the root definitions, UTopia injects the fuzz input according to the attributes analyzed by UTopia for the API parameters that receive their values from the root definition (static analysis of APIs determining the attributes is detailed in §III-B). For instance, in the constructor for the `Mat` class (line 18 in Figure 1), UTopia infers the *array↔length* relation, and assigns the size of `dim` (`Array` attribute) to the first argument (`ArrayLength` attribute) on line 18 and each element with fuzz input on line 17.

C. Unique Challenges of Utilizing Unit Tests

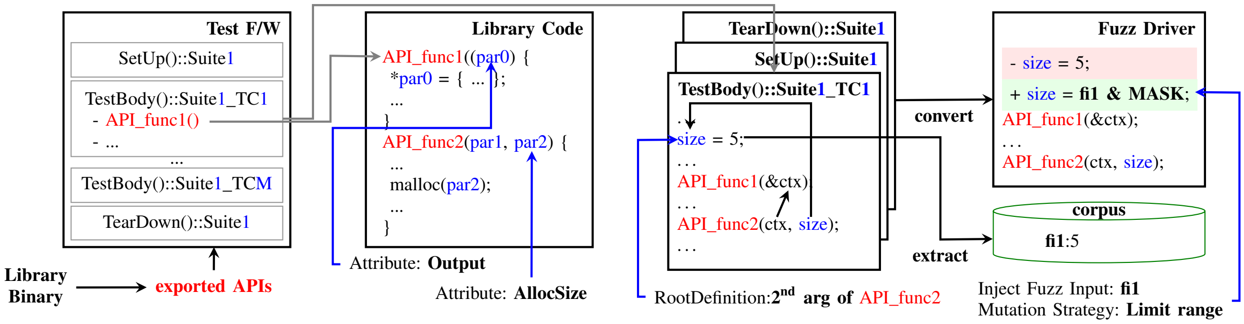
However, simply feeding the UTs as consumers to existing approaches would not appropriately work because the use of UTs introduces its own set of challenges (**UT-C**).

1) UT-C1: Analysis hindrance. As depicted in Figure 3, UT frameworks such as gtest are generally defined by complex class hierarchies intermixed with interfaces through which the user-defined test cases are indirectly invoked. However, static analysis, the primary tool for prior work [12, 16], suffers from imprecision in case of analysis across indirect calls which is further compounded by each consecutive call. The complexity of the class hierarchies increases the challenges to trace the control flows and identify the user-defined test code. Therefore, the drivers generated from UT by existing approaches may cause spurious crashes, requiring manual fixes before meaningful fuzz testing can be conducted. Alternatively, dynamic analysis [18] could manage the indirect calls, but it is unsuitable because of over-approximation and the difficulties in correlating semantics between argument values.

2) UT-C2: UT framework diversity. The diverse set of UT frameworks further amplifies the issues in UT-C1 as the aforementioned issues must be addressed according to the operation of each framework. If this is addressed in a labor intensive manner, such as manually fixing each generated driver, the wide applicability for UT would be infeasible.

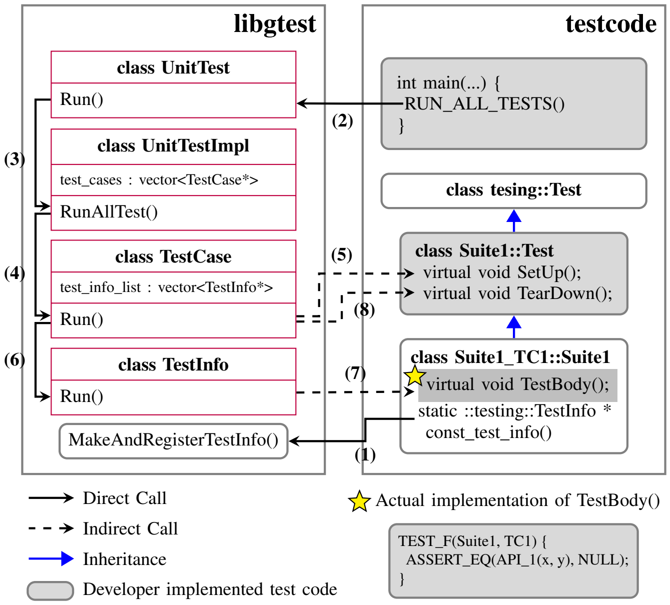
3) UT-C3: Assertion. Since the assertions in UTs not only check for critical states but may also be used to verify results against specific test values defined in UT, one must consider how they would affect fuzzing and handle the assertions appropriately as feeding fuzz input into arguments could easily trigger assertion conditions. If all assertions are ignored, the `nullptr` checks on pointers will make spurious crashes for dereferencing `nullptrs` much more likely. However, if all assertions are enforced, the test value checks will often block fuzz drivers from executing beyond the assertion statement.

Proposed approach. To surpass the challenges of working with UTs, UTopia supplements static analysis with an understanding of idioms used across the UT frameworks such as common setup and macro functions for individual test bodies. This enables UTopia to sidestep the challenges of analyzing complex relationships within the UT frameworks, such as following indirect calls, and also lightens the effort needed to accommodate new UT frameworks. Considering the duality of assertions, we provide the results of exploring several basic strategies to help developers make choices when generating fuzz drivers.



(a) UT F/W Structure Analysis (b) API Attribute Analysis (c) Fuzz Target Selection (d) Fuzz Driver Syntehsis

Figure 2: The workflow of UTOPIA to generate fuzz drivers.



```
TEST_F(Suite1, TC1) {
  ASSERT_EQ(API_1(x, y), NULL);
}
```

Figure 3: Hierarchy and Flow of Test Code. Simplified inheritance hierarchy of google test framework and the control flow of an implemented UT. Others including assertions and test statistics are omitted for brevity.

III. DESIGN

UTOPIA analyzes both UT and target library code to transform UT into effective fuzz drivers. The overall workflow of UTOPIA is illustrated in Figure 2. (a) UTOPIA utilizes the architectural nature of the UT framework so that it is only required to analyze developer implemented test functions instead of analyzing across the entire UT framework. (b) UTOPIA analyzes the library to identify attributes of API arguments. (c) Thereafter, UT analysis is performed to identify root definitions in which we can inject fuzz input without affecting valid API usage semantics. (d) Finally, driver synthesis is performed based on the analysis results.

In this section, for clarity, our explanations assume Clang and LLVM analyses, but our core ideas would be extendable to other similar analysis methods as well.

A. Structure Analysis of UT Frameworks

In general, UT frameworks provide APIs that allow users to define three functions for each test case: pre-test, test, and post-test. In case of googletest (gtest) in Figure 3, the UT framework exposes `SetUp()`, `TestBody()`, and `TearDown()` interfaces (respectively, pre-test, test, and post-test) to each test class. These functions implicitly ensure that 1) each test case is only reliant on those functions and 2) test cases are independent of each other. UTOPIA utilizes these features to construct valid API sequences by explicitly calling these functions in order within a fuzzing cycle to ensure the independence of each fuzzing cycle.

Clang AST Matchers. To locate these functions, UTOPIA utilizes clang AST Matchers to find functions with patterns on the abstract syntax tree (AST). For instance, in Figure 3, UTOPIA seeks a `CXXRecordDecl` with `testing::Test` class as a `CXXCtorInitializer` in its child nodes. Thereafter, `SetUp` can be found by searching a `CXXMethodDecl` with its name is `SetUp` within the found `CXXRecordDecl`. The other methods are similarly found. To support a new UT framework, developers only need to specify patterns for the test functions that the developers wrote, which minimizes engineering effort to support diverse UT frameworks.

B. API Attribute Analysis

UTOPIA considers all exported functions of a library as exposed APIs and analyzes each API argument to determine its attributes. UTOPIA performs inter-procedural analysis by leveraging def-use chains starting from API arguments to determine five attributes: *Output, FilePath, AllocSize, LoopCount,* and *Array↔Length(index)*.

Def-Use (DU) chain. As the attribute analysis focuses on argument-perspective internal behavior within the library, the analysis follows any uses of an argument along its DU chain, which connects definition and all uses (and uses of uses, thus, chain) reachable from that definition, to make sure whether the argument has certain properties. In addition, when a use is a value operand of a `store` instruction, the value and pointer operands are considered as if they have def-use relation to follow every possible use of the stored value. Notably, the `store` instruction in LLVM IR comprises value operand and pointer operand, suggesting that the value indicated by a value operand is stored into a pointer operand.

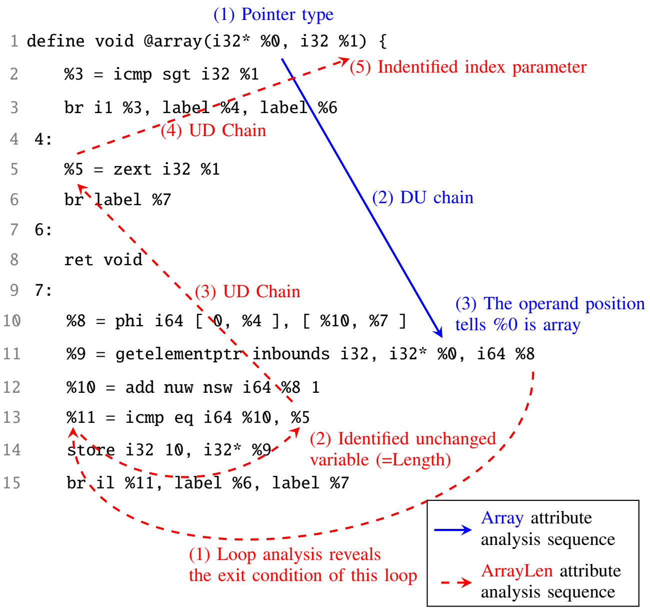
Inter-procedural analysis. UTopia basically performs analysis per function. If a use in a DU chain indicates an argument of a subroutine call, UTopia first resolves the analysis on the callee function and then merges the analysis results of the corresponding argument of the callee function. In case of external function calls, UTopia also supports the loading of pre-analyzed results of other libraries to obtain more precise results regarding external functions.

Analysis of Output. The output attribute indicates that an argument is used for outputting some value to the caller of the API. It is helpful to identify such parameters because fuzzing such parameters wastes the invested effort. In LLVM IR, the `store` instructions are used for writing value to memory and `load`/`gep` instructions are used for reading value from memory. Therefore, the output attribute can be conveniently identified by checking if the memory to which an argument points is only used by store instructions without any `load`/`gep` instructions.

Analysis of FilePath and AllocSize. The FilePath attribute represents arguments used as file paths in file operations, and the AllocSize attribute indicates arguments dictating the allocation size. It is useful for identifying FilePath, as one should fuzz the contents of the file indicated by the path rather than the file path string itself as fuzzing the string would mostly end in access errors. Upon identifying AllocSize, UTopia can abstain from inputting large fuzzing values that may trigger out-of-memory errors, which are not spurious crashes per se but are detrimental to the overall fuzzing run. As the processes required for identifying these attributes are mostly identical, we explain them at the same time. To identify them, we marked each property to well-known function parameters such as libc functions or compiler intrinsic functions. For instance, we mark `FilePath` attribute to the first parameter of `fopen` and mark `AllocSize` attribute to the first parameter of `malloc` or related compiler intrinsic functions. Thereafter, the marked attributes are propagated to any definition through inter-procedural analysis. To this end, if any uses of an API argument includes a marked parameter, the argument can be analyzed to contain attributes.

Analysis of LoopCount. The LoopCount attribute indicates that the argument determines the counter of a loop within the library. The identification of such arguments enables UTopia to avoid inputting large fuzzing values that may cause timeouts owing to repeating loops. If a compare instruction is detected within the DU chain of an argument, UTopia uses the LLVM LoopAnalysis to check whether the instruction is used as an exit condition of the loop. Upon detecting any condition, the argument can be deemed to contain the LoopCount attribute.

Analysis of Array↔Length. The Array and Length attributes each indicate arguments used as an array and the length of the array within library code, respectively. To identify these attributes, UTopia primarily utilizes `gep` instructions that access an memory with an index. A `gep` instruction comprises a base memory operand and an index operand. An argument is considered to bear the Array attribute if a use in its DU chain follows the base address of a `gep` instruction. After discovering an Array attribute, UTopia starts to analyze for the Length attribute only if 1) the `gep` instruction is in a loop and its index operand is an induction variable, 2) exit condition is present in the loop, and 3) exit condition remains unchanged throughout the loop. For example, the 10 in a `while(i<10)` statement would fulfill these requirements. If such an operand exists, UTopia starts to check whether the operand comes from another argument of the API by tracing its Use-Def (UD) chain, the reverse concept of DU chain. The analysis flow of UTopia is illustrated in Figure 4.



```
 1 define void @array(i32* %0, i32 %1) {
 2     %3 = icmp sgt i32 %1
 3     br i1 %3, label %4, label %6
 4  4:
 5     %5 = zext i32 %1
 6     br label %7
 7  6:
 8     ret void
 9  7:
10     %8 = phi i64 [ 0, %4 ], [ %10, %7 ]
11     %9 = getelementptr inbounds i32, i32* %0, i64 %8
12     %10 = add nuw nsw i64 %8 1
13     %11 = icmp eq i64 %10, %5
14     store i32 10, i32* %9
15     br il %11, label %6, label %7
```

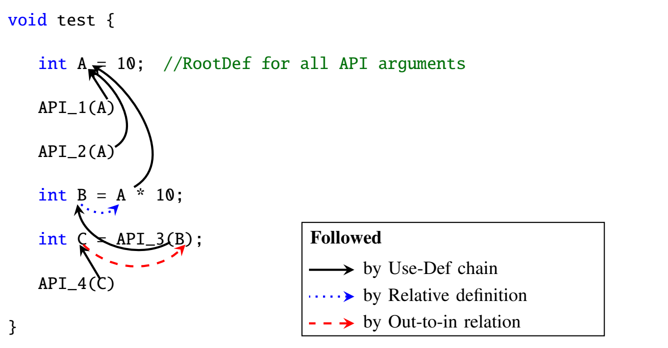
(1) Pointer type
(5) Indentified index parameter
(4) UD Chain
(2) DU chain
(3) UD Chain
(3) The operand position tells %0 is array
(2) Identified unchanged variable (=Length)
(1) Loop analysis reveals the exit condition of this loop
Array attribute analysis sequence
ArrayLen attribute analysis sequence

Figure 4: Analysis example. A simplified LLVM IR that has array and length arguments. LLVM loop analysis is used to identify that %8 is an induction variable in line 11 and the line 13 is the exit condition of the loop.

C. *Fuzz Target Selection*

This section describes the procedure followed by UTopia to determine *fuzz targets*, locations in which UTopia may insert fuzz input to appropriately fuzz library API call parameters (i.e., root definitions of parameters). In principle, this is basically done by finding the root definitions that define the values ultimately flowing into API parameters.

Root definition analysis. Root definition analysis is a backward data flow analysis to obtain definitions whose r-value is a constant value. Assuredly, constant values cannot be derived from any other variables that may be used in other statements in test code. Thus, the transformation of these constant values in root definitions enable UTopia to inject fuzz input without violating test code semantics. In particular, UTopia performs root definition analysis from all API arguments to gather every possible fuzz target candidate. An example of root definition analysis for all API parameters is presented in Figure 5. 'int A = 10' is the only identified root



```
void test {

   int A = 10;  //RootDef for all API arguments

   API_1(A)

   API_2(A)

   int B = A * 10;

   int C = API_3(B);

   API_4(C)

}
```

Figure 5: Analysis flow of root definition, Only ‘`int A = 10;`‘ is considered as valid fuzz input, which ensures the injection of fuzz input without violation of any API relations.

definition for all API arguments. The variations in the r-value of the root definition influence every API argument, while maintaining the relationships between the APIs. To determine all the possible definitions influencing the API parameters, the analysis is control flow sensitive and inter-procedural to find all possible definitions that can affect API parameters.

Inheritance of parameter attributes. To determine mutation strategies, UTOPIA must pair root definitions with the attributes of corresponding parameters. This is conducted by assigning parameter attributes to the root definitions that are used by the parameters directly. For instance, in Figure 5, Root definition ‘`int A = 10`‘ bears attributes of the first parameter of `API_1` and `API_2`. However, the attributes of the first parameter of `API_4` is not inherited, because the root definition is not directly used for that parameter. During the root definition analysis, the tracing target is changed from C to B by ‘`int C = API_3(B)`‘.

Inference of external functions. UTOPIA traces all input arguments if the tracing target is defined by external functions to find any possible definitions. UTOPIA utilizes output attributes analyzed by §III-B to identify input arguments only.

D. Fuzz Driver Synthesis

Fuzz input assignment. UTOPIA transforms each test case into a fuzz driver by replacing the identified fuzz targets with fuzz input assignment statements. Among the identified fuzz targets, UTOPIA excludes certain root definitions if its source code cannot be modified or an appropriate method cannot be determined for generating fuzz inputs. The criteria for exclusion is as follows:

- Root definitions in header files or out of project files
- Constants determined at compile time (e.g., sizeof(int))
- Assignments with return or output parameter of an external function (non-input parameter)
- Root definitions with `nullptr` assignment because we do not know how to initialize the object referenced by a pointer
- Function pointer parameters
- Values dependent on ignored values (e.g. ArrayLen of ignored Array)
- File proprieties (e.g., Read, Write)

After exclusion, UTOPIA replaces the r-values of the assignment statements with the fuzz input according to their data types and mutation strategies. UTOPIA adheres to the following mutation strategies to comply with API semantics:

- *FilePath:* Pass fuzz input to the content of a file instead of the file path.
- *AllocSize:* Limit the range of fuzz input for an argument used as memory allocation size.
- *LoopCount:* Limit the range of fuzz input for an argument used as an exit condition of a loop.
- *Array:* Fuzz as an array (i.e. create an array and assign fuzz input to each element of the array).
- *ArrayLength/Index:* Limit the fuzz input to one less than the size of the created array.

Fuzzing loop construction. UTOPIA constructs an entry function that is called once in each fuzzing loop. The entry function receives fuzz input from fuzzing engines (e.g., libfuzzer), and calls the identified and transformed test functions in order (e.g., `SetUp()`, `TestBody()` and `TearDown()` in gtest) to execute the fuzz driver with assigned fuzz input.

Initial seed extraction. Additionally, a simple yet effective procedure UTOPIA performs during UT analysis is to acquire the initial seed corpora embedded in the test code, which are the constant values in root definition statements identified as fuzz targets. These initial seeds allows the fuzz drivers to reach deep program state during the early stages of fuzzing and assists the fuzzers to expand their exploration into deep paths.

IV. Implementation

In this study, we implemented UTOPIA with 39K lines of code (excluding comments or blank lines), among which 37K LoC are C/C++ code to analyze libraries and unit tests and generate fuzz drivers, whereas the remaining 2K LoC are Python scripts to support and streamline the entire analysis and generation process. The code for analysis and fuzz driver synthesis leverages the analysis framework of LLVM/Clang and the conversion of the UT code into fuzz drivers is implemented with Clang ASTMatcher and Libtooling.

V. Evaluation

We evaluate UTOPIA by stating the following questions:

- **Automation.** How many UT-based projects can be automatically converted by UTOPIA, and how effective are the synthesized fuzz drivers compared to UT? (§V-A)
- **Fuzzing effectiveness.** How effective are the UTOPIA-generated fuzz drivers compared to the manually written drivers in terms of code coverage and bugs? (§V-B, §V-C)
- **Comparison.** How does UTOPIA compare with the existing approaches for the automatic fuzz-driver generation? (§V-E)
- **Design decisions.** How many spurious crashes are reduced, what is the best strategy for handling assertions, and how well UTOPIA analyzes API attributes? (§V-D)

Experimental setup. UTOPIA is separately evaluated with 25 OSS libraries from Table I that implement UTs with gtest or boost and 30 Tizen projects implementing the UT with Tizen TCT [5]. We made different settings for each experimental purpose, and the detailed settings are described in each subsection. In common, each fuzz driver ran as a single libFuzzer worker process in a Docker container installed with Ubuntu 18.04 base image. The container ran on Intel Xeon Gold 6258R processor (112 cores at 2.70GHz) and 251GB RAM.

A. Automatic Generation of Fuzz Drivers

UTopia could successfully synthesize the fuzz drivers from the test cases (TCs) for all 25 OSS projects that it attempted to generate without human intervention (listed in Table II).

Setup. We selected six popular projects from Github, 11 included in the benchmark suite of `ossfuzz`, seven external libraries that Android relies on, and one external project used by Tizen. They represent projects with various sizes (a few thousands to a few millions LoC), building systems (e.g., `cmake`, `bazel`, and `ninja`), and unit testing framework (e.g., gtest and boost). For all 25 projects, each UTopia-generated fuzz driver ran for 1 per-core hour.

Generated fuzz drivers. Among 5,523 TCs in the projects, we excluded 1,039 TCs implemented with the macro functions not handled in our prototype implementation, i.e., those other than `TEST` and `TEST_F` (for gtest) or `BOOST_AUTO_TEST_CASE_FIXTURE` (for boost) (`Oths.` in Table II). For the remaining 4,484 TCs, UTopia removed 1,769 TCs (39% of the inspected 4,484 TCs) according to our exclusion criteria in the process of determining root definitions. In total, UTopia automatically produced all 2,715 fuzz drivers (100%) from the feasible candidate TCs in these projects. For interested readers, we share the ratio of characteristics within the removed TCs in Table VIII of the Appendix.

The generated fuzz drivers included 2,292 functions that the libraries export, and directly provided the fuzz input to 56% of the used functions in libraries, implying that the other half of the library functions are utilized to construct a proper preliminary state for testing.

vs. unit testing. The UTopia-generated fuzz drivers increase the unique testing coverage of these projects by 1.4-58.2% in comparison to the corresponding TCs in one per-core hour of fuzzing execution of each generated driver. The coverage of each fuzzer is significantly increased up to 60 times (column `MG` in Table II). These findings highlight two interesting aspects of UTopia: 1) automatically expanding the testing capability of unit tests and 2) TCs are designed to check what developers expected to be correct, but fuzzers focus on what they didn't expect. Both approaches are nicely bridged by UTopia.

Generation time. It took a total of 15.7 per-core hours to analyze 4,484 TCs and libraries and generate 2,715 fuzz drivers from them, suggesting that we can produce a piece of fuzz driver source code every 15 per-core seconds. More specifically, it took 15.6 per-core hours to analyze and 6 per-core mins to synthesize the corresponding fuzz drivers, yet it is negligible in comparison to their building time (2 hours on our 112-core machine).

Crashes. We manually reviewed a total of 1,167 unique crashes that occurred during the fuzzing with this setup. Among the unique crashes, i.e., groups of crashes distinguished by the uniqueness of stack traces included in the crash reports, 109 were bugs, 572 were normal crashes due to `timeout`, `oom`, and `abort`, and 486 were spurious crashes caused by API misuse. The normal crashes occurred even if the correct API usage is respected, and degraded the fuzzing execution performance. 75% of the normal crashes resulted from `timeout`, mostly because the testing of such APIs involved a long process duration (e.g., image data processing). 43% of spurious crashes were caused by the attributes that UTopia does not cover and 33% were caused by analysis failure. However, no spurious crash was caused by the incorrect sequences. The detailed statistics of spurious crashes are stated in Table X in the Appendix.

B. Fuzzing effectiveness

The effectiveness of fuzzers can be directly explained by the number of discovered bugs. UTopia discovered a total of 123 bugs: 109 bugs in a short period of trial run for a few days with the generated 2,715 fuzz drivers for 25 OSS projects, and 14 bugs from about two weeks with 2,411 fuzz drivers for 30 Tizen native libraries.

Note that UTopia used exactly the same API sequences in TCs but still discovered new bugs, some of which have existed for years. This illustrates the benefit of UTopia's approach: leveraging TCs to find new classes of bugs that the developers missed to capture during testing.

Setup 1 (25 OSS projects). The evaluation is based on the results from §V-A & Table II.

Setup 2 (30 Tizen projects). We generated the fuzz driver source code with UTopia for 30 projects from Tizen which were adopted by the Tizen community [4]. Due to their internal policy, we could not share the details of the drivers and only evaluated their fuzzing results.

Tizen evaluation. The fuzzing run for the Tizen libraries produced fewer bugs than the shorter trial run for the OSS libraries, primarily because Tizen performs rigorous inspection based on the static analysis on the new code commit. Even so, UTopia found 14 confirmed bugs in 11 of the 30 Tizen libraries tested, some of which had been latent for up to 7 years. All bugs were fixed prior to the version 6.5 release of Tizen.

Table IX in the Appendix lists the details of the Tizen bugs. It is notable that, despite the project policy requiring developers to test their own code with at least 80% of test coverage, including tests for error cases, most of the bugs found in Tizen resulted from missing validations, such as `nullptr` checks and return checks. As test code are susceptible to developer's bias, UTopia can act as a good complement as it can fuzz test what the developers may have overlooked.

Target Library					Unit Tests				UTopia-Generated Fuzz Drivers									
Name	SR	LoC	BS	#eFn	TF	Cov.	TcCov.	#TC	Analyzed Test Cases			Functions		Time(sec)		Cov./UCov	AG/MG	#UC (S/N/B)
									#Oths.	#Ign.	#Gen.	#tested	#w/input	AT	GT			
Node.js	O	3M	gn	3,065	G	11.8%	11.5%	124	0	83	41 (100%)	63	24	1,158	21	13.6%/ 3.1%	1.2/ 2.0	7 (4/1/2)
libaom	O	363K	cm	5,065	G	51.5%	48.4%	682	531	36	115 (100%)	109	98	18,290	7	54.6%/ 6.2%	1.0/ 37.5	65 (27/18/20)
assimp	O	356K	gn	5,055	G	45.5%	23.9%	449	0	199	250 (100%)	96	56	1,019	7	36.3%/ 13.9%	1.3/ 6.5	133 (35/42/56)
libvpx	O	248K	cm	1,446	G	47.6%	27.7%	373	315	18	40 (100%)	42	32	4,522	6	34.3%/ 6.5%	1.1/ 2.0	28 (18/8/2)
tesseract-ocr	O	158K	cm	3,650	G	62.4%	57.1%	477	165	72	240 (100%)	339	187	2,294	66	59.9%/ 2.9%	1.0/ 3.2	138 (81/48/9)
openh264	O	92K	cm	1,523	G	61.3%	33.6%	320	21	186	113 (100%)	133	94	4,521	8	34.3%/ 0.8%	1.0/ 1.3	82 (63/13/6)
libphonenumber	O	53K	cm	510	G	65.2%	63.7%	324	0	78	246 (100%)	152	97	4,675	9	65.2%/ 2.3%	1.0/ 3.0	50 (19/29/2)
wabt	O	47K	cm	1,034	G	24.9%	24.6%	190	0	110	80 (100%)	61	40	430	2	26.3%/ 1.8%	1.1/ 2.3	65 (27/36/2)
leveldb	O	21K	cm	397	G	87.1%	86.0%	218	0	43	175 (100%)	92	51	1,001	12	85.3%/ 1.0%	1.0/ 4.0	87 (25/61/1)
libhtp	O	20K	gn	386	G	73.6%	73.3%	339	3	0	336 (100%)	191	141	490	99	78.0%/ 5.8%	1.1/ 4.9	52 (21/29/2)
jsonnet	O	13K	cm	98	G	35.9%	35.9%	45	0	0	45 (100%)	6	4	16	2	41.3%/ 5.3%	1.2/ 14.9	3 (3/0/0)
uriparser	G	8K	cm	42	G	90.5%	88.7%	92	0	10	82 (100%)	50	50	80	2	92.1%/ 5.1%	1.0/ 14.2	10 (8/0/2)
mediapipe	G	225K	bz	2,237	G	47.0%	37.5%	524	4	321	199 (100%)	226	66	6,787	12	38.7%/ 1.3%	1.1/ 1.6	187 (57/130/0)
filament	G	64K	nj	5,948	G	32.8%	25.0%	292	0	170	122 (100%)	219	92	4,576	20	27.7%/ 2.7%	1.1/ 3.4	116 (19/97/0)
muduo	G	16K	cm	359	B	15.3%	9.9%	30	0	25	5 (100%)	11	5	32	1	11.0%/ 1.1%	1.3/ 1.3	0
vowpal_wabbit	G	81K	cm	1,383	B	20.3%	14.8%	224	0	155	69 (100%)	64	46	1,968	3	16.4%/ 1.6%	1.1/ 1.5	39 (23/13/3)
ledger	G	51K	cm	32	B	9.3%	9.3%	17	0	0	17 (100%)	32	10	410	1	10.5%/ 1.1%	1.2/ 1.6	11 (2/8/1)
cpuinfo	A	423K	cm	66	G	54.2%	15.6%	142	0	136	6 (100%)	1	1	2,068	6	16.2%/ 0.6%	1.0/ 2.3	1 (1/0/0)
minijail	A	16K	gn	162	G	54.1%	47.9%	189	0	30	159 (100%)	102	60	767	31	39.6%/ 1.3%	1.2/ 22.0	5 (4/1/0)
pthreadpool	A	12K	cm	54	G	69.3%	69.3%	291	0	1	290 (100%)	24	24	317	4	74.7%/ 5.4%	1.1/ 4.5	0
cpu_features	A	6K	cm	36	G	51.5%	9.43%	31	0	27	4 (100%)	5	3	23	11	9.6%/ 0.2%	1.2/ 1.7	0
puffin	A	5K	cm	92	G	83.6%	66.3%	44	0	17	27 (100%)	34	21	122	1	71.5%/ 5.2%	1.1/ 60.6	28 (19/8/1)
bsdiff	A	4K	gn	137	G	57.2%	43.4%	66	0	42	24(100%)	34	19	207	11	44.3%/ 2.0%	1.1/ 4.2	10 (8/2/0)
sfntly	A	23K	cm	897	G	48.7%	48.2%	23	0	7	16 (100%)	192	44	431	2	49.3%/ 1.4%	1.1/ 1.1	31 (13/18/0)
snappy	T	6K	gn	46	G	75.9%	75.9%	17	0	3	14 (100%)	14	12	112	1	79.5%/ 3.7%	1.1/ 2.3	19 (9/10/0)
Total	-	53M	-	33,720	-	-	-	5,523	1,039	1,769	2,715 (100%)	2,292	1,277	15.6hr	6min	-	-	1,167 (486/572/109)

Table II: Projects used for evaluation and results for UTopia-generated fuzz drivers. **Target Library: SR** = Source repository (O:OSSFuzz / G:GitHub / A:Android / T:Tizen), **BS** = Build system (cm:cmake / gn:gnu make / nj:ninja / bz:bazel), **eFn** = Exported functions in a library, **Unit Tests: TF** = Testing framework (G:gtest / B:boost), **TcCov.** = Region coverage of the target library with the test cases that fuzz drivers are generated from (region coverage: measured with a tool from clang++ [9]), **TC** = Total number of test cases, **UTopia-Generated Fuzz Drivers: Oths.** = TCs implemented with macro functions other than TEST, TEST_F or BOOST_AUTO_TEST_CASE_FIXTURE, **Ign.** = TCs ignored by UTopia based on the exclusion criteria, **AT** = Per-core time to analyze library and unit test code, **GT** = Per-core time to generate source code, **UCov** = Unique coverage of UTopia compared with TC Cov., **AG** = The ratio of the coverage with the aggregation of unique regions across all fuzzers to that of TCs from which the fuzzers were made, **MG** = The individual maximum growth ratio of a single fuzzer compared to execution with the initial seed, **UC** = Total unique crash (unique crashes: crash groups identified by uniqueness of stack traces included in crash reports), **S** = Spurious unique crashes due to API misuses, **N** = Normal crashes with timeout/oom/abort, **B** = Crashes reported after manual review.

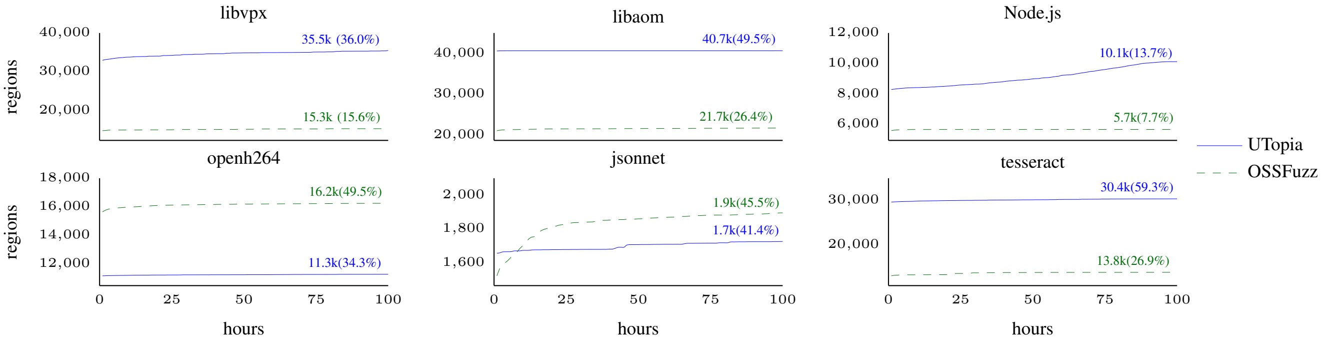


Figure 6: Coverage over 100 per-core hours of fuzzing UTopia generated drivers and OSS-Fuzz drivers. The graphs show the average and 95% confidence interval of the coverage increase of 10 repeated fuzzing runs (measured in regions [9]). The total fuzzing resource consumed by each approach for each library is matched (i.e., each fuzz driver is allotted 100/number_of_drivers hours on a single CPU). The details on the drivers of each approach is reported in Table III.

OSS evaluation. UTopia discovered 109 bugs in 14 projects among 25 OSS projects (see, Table II). We manually analyzed all unique crashes found by UTopia, and reported them to the maintainers after review. Of the 109 bugs reported so far, 56 were confirmed or fixed by the maintainers. Similar to Tizen, these bugs were missed because of the existing areas overlooked by TCs within the input space of their API.

Fuzzing internal APIs. UT code often directly tests the internal APIs of libraries as developers may want to assure the correct operation of library internals. UTopia can naturally enable fuzz testing of these internals by transforming such UT code. However, depending on the API, the bugs found from such fuzzing may not be regarded as meaningful. In particular, the project maintainers did not accept 5 out of 109 reported bugs in the 25 projects, because the crashes were considered non-security issues. For instance, UTopia detected a bug in a

Library	OSS-Fuzz		UTopia		
	#Drivers (APIs)	Coverage	#Drivers (APIs)	Coverage	Unique Coverage
libvpx	2 (7)	15.6%	40 (43)	36.0%	23.4%
libaom	1 (5)	26.4%	115 (109)	49.5%	27.8%
Node.js	2 (32)	7.7%	42 (60)	13.7%	6.2%
openh264	1 (7)	49.5%	113 (132)	34.3%	10.3%
jsonnet	1 (5)	45.5%	45 (6)	41.4%	3.2%
tesseract	1 (9)	26.9%	240 (356)	59.3%	33.6%

Table III: Comparison of UTopia and OSS-Fuzz drivers. Results are averaged over 10 fuzzing runs of 100 per-core hours.

Project	Approach	API Sequence
	(Coverage)	Original UT or Driver name
openh264	OSS-Fuzz	`WelsCreateDecoder` → `Initialize` → `SetOption` → `DecodeFrameNoDelay` → `Uninitialize` → `WelsDestroyDecoder`
	(48.4%)	`decoder_fuzzer`
	UTopia	`WelsCreateDecoder` → `Initialize` → `SetOption` → `WelsDecodeBs` → `Uninitialize` → `WelsDestroyDecoder`
	(44.4%)	`DecoderParseSyntaxTest-DecoderParseSyntaxTestAll`
jsonnet	OSS-Fuzz	(Note: We use `jsn` for short form of `jsonnet`.) `jsn_make` → **`jsn_import_callback`** → `jsn_evaluate_snippet` → `jsn_realloc` → `jsn_destroy`
	(46.1%)	`convert_jsonnet_fuzzer`
	UTopia	`jsn_make` → `jsn_evaluate_snippet` → `jsn_realloc` → `jsn_destroy`
	(40.6%)	`JsonnetTest-TestEvaluateSnippet`

The red underlined text highlights different APIs between paired sequences and the blue bold text shows additional APIs in a sequence.

Table IV: API sequences and coverage (24 cpu-hours) for driver pairs with similar API sequences from OSS-Fuzz and UTopia.

libnode API, a C++ implementation of Node.js. However, the API was not considered as an attack surface, because it was not reachable from the interface of Node.js script as detailed in §-A in the Appendix. Nonetheless, such findings can be meaningful toward the reliability and robustness of a library. Certain maintainers, such as those of Node.js and tesseract, advised us to submit the patches to improve the robustness of their implementation[1] and accepted our patches.

C. Comparison to OSS-Fuzz drivers

We compared the sets of the fuzz drivers generated by UTopia to the manually written fuzzers for six OSS-Fuzz projects and shared the results in Table III and Figure 6. For interested readers, the coverage comparison results of 24 hour fuzzing runs of the remaining seven OSS-Fuzz projects are listed in Table XI in the Appendix. As our initial tests with these remaining seven did not provide any distinct insight than the six projects we cover below, we omitted their details for brevity.

Setup 1. For this evaluation, we examined the achieved code coverage of OSS-Fuzz and UTopia for over 100 hours of fuzzing, showing the effectiveness of the automatic approach of UTopia in comparison with hand-tuned, best performing fuzzers. Note that the UTopia driver sets for the libraries were the same as those listed in Table II and that we matched the total consumed resource for each approach to 100 per-core hours (e.g., for Node.js, OSS-Fuzz's two drivers each ran for 50 hours on single CPUs, while UTopia's 41 drivers each ran for about 2.4 hours on single CPUs). For the OSS-Fuzz drivers, we also followed the fuzzing scripts found in the OSS-Fuzz project in order to run their drivers at their full potential (e.g., providing their intended initial corpora and keyword dictionaries when available). The coverage for plotting Figure 6 is aggregated by time (e.g. libvpx has 40 generated drivers, so we aggregated the results from generated inputs of the first 1.5 seconds of each driver, totaling at one minute, for the results of the first minute of fuzzing the set of drivers).

Code coverage. In regards of code coverage, UTopia's fuzz drivers outperformed 4 out of 6 projects by an average of 20.5%, and underperformed (average 9.7%) on 2 projects. The coverage gain of UTopia could be considered mainly due to it fuzzing more number of APIs. Even so, in Figure 6, we can observe a continuous increase in UTopia's coverage over time, indicating that UTopia's large number of tested APIs provide, beyond their initial impact on coverage, valid opportunities in further exploring each library. One interesting point is that, as observed in Table III, even when UTopia underperformed in terms of total explored coverage, it still explores unique regions in the library that are not reached by the OSS-Fuzz drivers.

Taking a deeper look into generated drivers. To compare the quality of UTopia generated drivers to that of manually written ones, we have searched for drivers with matching API sequences. Though there were no exact matches, two pairs of drivers had close resemblance. The API sequence of each driver and the coverage after 24 hours of fuzzing is given in Table IV. As observed, UTopia slightly underperformed in comparison to the OSS-Fuzz drivers in terms of coverage. The reason behind this phenomenon is the same as why UTopia underperformed in the overall coverage of openh264 and jsonnet as well: UTopia provided fuzz input to additional API arguments which, in these cases, contribute little to coverage exploration.

Setup 2. Drivers of OSS-Fuzz and UTopia with comparable API sequences to each other are run for 10 trials of 24 per-core hours per fuzz driver (Table IV).

For openh264, the primary reason for this is that several UTs mocked the internal logic of the existing APIs to directly test the inner APIs without initializing the objects required for calling the outer API. This can impair UTopia drivers as UTopia may detect the mocked assignments as fuzz targets for the internal APIs and attempt to fuzz them. It would miss out on exploring other APIs that are supposed to be called alongside the inner API when the outer API is called. For example, in our compared drivers (Table IV), `WelsDecodeBs()` called in the UTopia driver is actually one of the APIs called within the `DecodeFrameNoDelay()` API called from the OSS-Fuzz driver. In the case of jsonnet, the fundamental reason is that jsonnet APIs accept a string argument which they mainly use for error logs. While the OSS-Fuzz driver is optimally written to ignore this argument, the automatically generated UTopia drivers have no way of determining the usefulness of fuzzing such arguments and end up putting effort in fuzzing the string as well as the argument fuzzed in the OSS-Fuzz

[1] https://github.com/tesseract-ocr/tesseract/issues/3591

driver. In our compared drivers, `jsonnet_evaluate_snipper()` accepts such an argument which is given an empty string by the OSS-Fuzz driver and fuzz input by the UTopia driver.

Note that, though not as optimal as the OSS-Fuzz drivers, UTopia still successfully located and fuzzed the same arguments that were ultimately fuzzed by OSS-Fuzz, implying that the automatically generated drivers are capable of providing meaningful fuzz tests.

API coverage. Additionally, we have investigated how many APIs found in the handcrafted fuzzers were covered by UTopia because APIs selected for manual fuzzers can be thought as what developers were interested in testing. Among the 65 APIs in the manual fuzzers, UTopia covered 60. It means 92% of interesting APIs were covered by UTopia while it also covers an additional 646 APIs in test code. Even though UTopia did not cover all the interesting APIs, its overall coverage was mostly higher than manual drivers and it could explore many unique regions of code owing to the opportunities found in fuzzing the additional APIs.

D. *Evaluating* UTopia*'s Design Decisions*

Setup. Generated drivers of UTopia satisfying certain criteria (according to each evaluation) were executed for 10 trials of 24 per-core hours per fuzz driver (Table V) or 12 per-core hours per fuzz driver (Figure 7).

Mitigating undesirable crashes. We experimented to investigate the influence of the analyzed attributes ArrayLength, AllocSize, and LoopCount, which were obtained through library analysis, on the reduction of spurious crashes and crashes caused by the detrimental fuzz input. For evaluation, we selected fuzz drivers from three projects that tested the API arguments with the three attributes by removing one of the properties for comparison. As listed in Table V, the settings without either the ArrayLength or AllocSize attribute resulted in a drastic increase in crashes by up to two orders of magnitude with marginal increase in coverage. On the other hand, without the LoopCount attribute, no differences could be observed in crashes, but the exec/sec performance degraded significantly up to 40%. For the assimp project, the coverage and exec/sec were reduced to 37% and 2%, respectively compared to including the attribute, when the AllocSize attribute was removed. In case of the libhtp project, the crash increased by 645 times without the ArrayLength attribute with poor coverage and exec/sec performance. Moreover, the omission of the LoopCount in leveldb reduced the exec/sec performance to 41%.

Assertion handling. To improve the performance of fuzzing, UTopia utilizes the semantic idioms of UT's in generating fuzzing drivers. More specifically, we evaluated seven strategies for handling UT assertions: i) *early termination* that rejects to proceed further execution on the condition of check failure, ii) *nop* that ignores all assertions, iii-v) *nopNs* that are derivatives of nop), vi) *nullptr check only* that maintains only assertions checking for `nullptr`, and vii) *dependent object check only* that maintains assertions checking for objects on which the subsequent APIs are dependent. As depicted in Figure 7, the fuzzing results of three libraries exhibited the most significant deviation between the strategies. In particular, the fuzz drivers for `libhtp` and `assimp` had many assertions on `nullptr` check, and the fuzz drivers for `uriparser` had many assertions to compare the return values of functions to specific values. Overall, ignoring assertion logic is detrimental to fuzzing performance, because both coverage and execution per time suffers significantly. Although maintaining only `nullptr` checks could be more beneficial in certain cases, the other strategies displayed comparable performance in terms of coverage or better performance in terms of execution.

Statistics of attribute analysis. We evaluated the adequacy of the API attribute analysis of UTopia for identifying the interdependencies between the parameters of an API and the impact of individual parameters on the internal behavior of the API. Accordingly, we selected the top-most projects (assimp, libhtp, leveldb) with such attributes that are used for crash mitigation evaluation; the remaining projects briefed in Table VI were randomly selected. The accuracy of the attribute analysis is detailed in Table VI. All the analyzed attributes were correctly identified (nearly 100% of precision), but UTopia missed to identify half of the ArrayLeng/Index attributes (45% of recall). These false negatives were estimated to account for 33% of the aforementioned spurious crashes in §V-A.

E. *Issues in Comparison to existing tools*

The most appropriate method for comparing the merits of UTopia against prior work would have been to generate fuzz drivers using the same consumer. Unfortunately, neither Fudge [12] nor Intelligen [29] is publicly available. Although it is reported that the fuzz drivers originally generated by Fudge made their way into OSS-Fuzz, aside from what was reported in their paper, there is no public source that could be reviewed to determine which OSS-Fuzz fuzzer had benefited from Fudge. FuzzBuilder [17] could not be used to perform a fair comparison as the quality of its drivers heavily depend on manual configurations to specify unit test functions, and target API arguments. FuzzGen [16] opened up its code but due to its PoC nature, its generated drivers for UT code could not offer a meaningful comparison despite our efforts in performing post-generation manual fixes to make the drivers work. The main issues that prevented FuzzGen from generating workable drivers from the UT code was 1) analysis limitations discussed in §II-A and §II-C, and 2) FuzzGen's implementation decision to only rely on primitive types and library defined compound types prevents it from properly navigating through UT defined classes which breaks its analysis and leaves incomplete fuzz drivers. We share the details of our experience with the FuzzGen PoC code in §-B in the Appendix. Alternatively, we could compare similar UTopia drivers against publicly available drivers of previous work as we conducted earlier against OSS-Fuzz in Table IV. Nonetheless, we could not find any combination of UTopia drivers that would provide meaningful comparison (similar API sequences) against the public drivers.

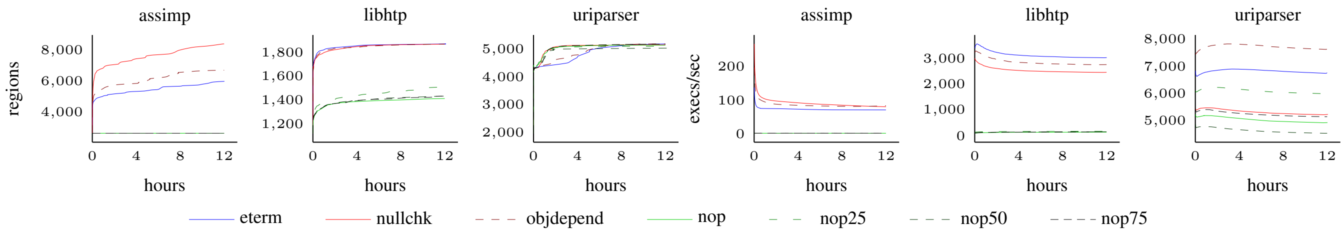


Figure 7: Comparison by strategies with handling assertions: i) early termination (maintain all asserts), ii) no operation (ignore all asserts), iii-v) nopN (N percent of nop operation from the beginning of test sequence), vi) `nullptr` check only (maintain asserts that check `nullptr`), and vii) dependent object check only (maintain asserts that check objects on which the subsequent APIs are dependent). A single fuzz driver per project was selected and ran 10 times for 12 per-core hours.

Project	**Region Coverage**				**Crash Count**				**Execution Count**			
	ALL	**w/o Loop**	**w/o Alloc**	**w/o Arr**	**ALL**	**w/o Loop**	**w/o Alloc**	**w/o Arr**	**ALL**	**w/o Loop**	**w/o Alloc**	**w/o Arr**
assimp	174 (100%)	160 (92%)	**64** **(37%)**	174 (100%)	114,790 (1x)	141,859 (1.2x)	**641,105** **(5.6x)**	**355,891** **(3.1x)**	6,544 (100%)	7,098 (108%)	**126** **(2%)**	**179** **(3%)**
libhtp	2,719 (100%)	2,719 (100%)	2,709 (100%)	**1,236** **(91%)**	2,298 (1x)	2,267 (1x)	**413,048** **(180x)**	**1,480,745** **(645x)**	48,672 (100%)	26,505 (105%)	40,088 (82%)	**51,346** **(54%)**
leveldb	144 (100%)	144 (100%)	**118** **(82%)**	144 (100%)	0 -	3,181 -	**462,948** -	**375,085** -	31,492 (100%)	**12,979** **(41%)**	31,615 (100%)	**18,715** **(59%)**

Table V: Crashes, coverage, and execution performance of fuzzing with three attributes (all), without AllocSize (w/o Alloc), without ArrayLength (w/o Arr), and without LoopCount (w/o Loop) attribute. Each fuzzer ran 10 times for 24 per-core hours without an initial seed corpus to eliminate the seed effects.

Project	**FilePath**			**AllocSize**			**LoopCount**			**Array**			**ArrayLength**		
	G	**P**	**R**	**G**	**P**	**R**	**G**	**P**	**R**	**G**	**P**	**R**	**G**	**P**	**R**
libhtp	-	-	-	27	.96	1.0	10	1.0	1.0	48	1.0	.88	42	1.0	.41
assimp	2	1.0	.5	4	1.0	1.0	3	1.0	1.0	34	1.0	.94	3	1.0	1.0
jsonnet	-	-	-	4	1.0	1.0	-	-	-	3	1.0	.67	-	-	-
Node.js	-	-	-	2	1.0	1.0	1	-	.0	5	1.0	.4	3	1.0	.33
snappy	-	-	-	-	-	-	-	-	-	10	1.0	.5	8	1.0	.38
ledger	-	-	-	1	1.0	1.0	1	1.0	1.0	2	1.0	1.0	1	-	.0
leveldb	3	-	.0	2	1.0	1.0	3	1.0	1.0	4	1.0	1.0	2	1.0	1.0
libphonenumber	-	-	-	-	-	-	7	1.0	1.0	4	1.0	.75	3	1.0	.67
Total	5	1.0	.2	40	.98	1.0	25	1.0	.96	110	1.0	.84	62	1.0	.45

G: Ground truth, **P**: Precision, **R**: Recall

Table VI: Precision and recall of analysis for each attribute.

VI. Discussion & Limitation

A. *Remaining Sources of Spurious Crashes*

For practical purposes, a fuzz driver should be able to strike a balance between state exploration (i.e., input mutation) and avoiding spurious crashes (i.e., respecting UTs). As both these goals are not easily complementary, UTopia made several design decisions based on our experiments; however, there is still scope for improvement:

Non-conventional relations. Our analysis can recognize five common relation and several argument types in §III-B and §III-C, but non-conventional or highly customized usages can not be understood for generating fuzz drivers. For instance, a char* type can be used as a format specifier in `printf()`-like functions, but UTopia would likely select the parameter as a fuzz input along with the previously used strings as a seed corpus. Any mutated inputs that include additional format specifiers like `%n` would yield in spurious crashes.

Insufficient error handling. As the UT code is used only for testing, developers tend to hard-code non-essential parameters and provide a fewer number of checks for error handling unlike production code. For instance, developers commonly skip checks for correct construction or proper allocation of an object, which is sensible in the context of UTs using static parameters. However, if such a UT becomes the basis for a fuzz driver, any erroneous cases would lead to spurious crashes. Although UTopia strives hard to recognize such cases (e.g., embracing `nullptr` checks), several observed cases could have been avoided if an experienced developer provided a proper check for handling errors in UTs.

B. *Limitation in* UTopia*'s Analysis*

Root definition for file paths. In certain test cases, the file path strings are created through multiple string operations. In this case, if UTopia creates a file for fuzzing and assigns its path at the root definition of the string (prior to all the operations), the actual path accessed by an API would be incorrect. To avoid this, UTopia heuristically assigns the generated fuzz file path to the closest string assignment/operation preceding the API. However, due to this heuristic, UTopia could fail in reflecting the original UT logic in the generated fuzz driver as the file path assignment location is not a root definition.

Constant value aliases in UT logic. When test cases directly use constant values instead of variables, UTopia might struggle to generate suitable drivers. For instance, let us assume a simple test case: `int i=5; ASSERT_EQ(decode(encode(i)),5); API(i);`. UTopia would generate a driver by changing the assignment to `int i=fi;` but it will be unable to test the `API()` with values other than `fi=5` due to the assertion. We would want to ignore the assertion in this case but, as we have evaluated in §V-D, we cannot simply opt to automatically ignore assertions as it could deteriorate the fuzzing performance. Another approach might be to handle same valued variables and constants as aliases and change the test code as follows: `int i=fi; ASSERT_EQ(decode(encode(i)),fi); API(i);`. But this would require additional careful analysis as values coincidentally sharing the same value being treated as aliases can lead to a new separate class of spurious crashes (e.g., in the above case, if an unrelated array is accessed by `arr[5]`, aliasing all constant `5`s with `i` and fuzzing them could lead to an out-of-bounds access). We leave analyzing and handling such cases as future work.

VII. Related Work

UTopia is closely related to recent studies conducted on fuzzing. Recently, several state-of-the-art fuzzers have been proposed in the literature to effectively and efficiently detect bugs. AFL [27] is a representative feedback-guided, grey-box fuzzer. To get feedback, it measures the achieved edge coverage via instrumentation. The feedback is used to evaluate how good an input is. It scores each input to reuse the best one for the next run, and mutate the selected input based on the genetic algorithm. This process is iterative and this simple strategy works very effectively. Therefore, several improvements [8, 11, 13–15, 19–22, 22–24, 26] have been introduced to date on seed scheduling, input mutation, and feedback mechanism. However, most have focused on fuzzing programs that use files or commands as input [10].

Library fuzzing. Unlike end-to-end fuzzing, library fuzzing requires in-depth knowledge of a target library such as correct API usage and prerequisites for API calls when implementing fuzz drivers that deliver fuzz inputs from a fuzzing engine to API arguments. OSS-Fuzz [2], which mainly employs libFuzzer [25], provides rewards to open source maintainers when they implement fuzz drivers [3]. However, as fuzz drivers typically require manual efforts, many open source projects are yet to adopt fuzzing. FuzzBuilder [17], was proposed to partially alleviate such manual effort by proposing a tool that helps transforming UT into fuzz drivers. Though the focus on UT conversion is similar to UTopia, as it requires manual configuration to specify test functions and target API parameters to generate fuzz drivers, the quality of the resulting drivers heavily depend on the manual work. On the other hand, UTopia enables automatic and reliable generation of fuzz drivers by analyzing UTs, and aims to accelerate the adoption of library fuzzing at a larger scale.

Automatic fuzz driver generation. A few automated fuzz driver generation approaches have been proposed recently. Fudge [12] looks for buffer access `(uint8_t* data, uint32_t size)` parameter signatures and extracts dependent code lines to compose fuzz drivers. FuzzGen [16] statically analyzes API dependency from consumer code and merges them to have a long API call sequence for a fuzz driver. Both work infer API usage mainly from the consumer side, leading to invalid or less efficient fuzz drivers (i.e., a higher chance of triggering spurious crashes). Hence, Fudge needs humans in the loop to evaluate and update fuzz drivers, and FuzzGen requires manual review process to repair the generated drivers. Intelligen [29] selects a function that has potentially dangerous operations (e.g., memory/pointer access) to synthesize fuzz drivers. However, as it fails to consider API relations, the generation fuzz drivers are far from complete. Instead of inferring API relations, UTopia directly uses the carved API call sequence from UT that is carefully written for testing. This approach can greatly reduce the incorrectness of synthesized fuzz drivers. As a result, UTopia's approach is much reliable and so applicable to a wide range of libraries in various scales (see Table I).

Unlike these static approaches that analyze source code, there exist two projects that attempt to infer API relation in a dynamic manner. WINNIE [18] aims to fuzz closed-source libraries on Windows via automatic fuzz driver generation and fast-cloning of Windows processes. APICraft [28] targets closed-source libraries of MacOS SDK. Both directly trace API sequences during runtime by executing the target programs (e.g., from manual dry run) and then, reuse the observed API call sequences in generating new fuzz drivers. These dynamic approaches are fundamentally inappropriate for large scale adoption and push-button automation UTopia aims to achieve.

VIII. Conclusion

In this paper, we propose UTopia that automatically generates fuzz drivers from available unit tests with no or minimal human effort. It not only understands the semantic constructs of the unit test frameworks, but also analyzes the implementation of each library's APIs under testing. As a result, UTopia is able to produce numerous fuzz drivers with valid API call sequences in a scalable manner. We show that UTopia can be widely applicable by showing that UTopia successfully generates fuzz drivers for 55 popular open-source projects. Our evaluations shows that UTopia can achieve more code coverage (average of 20.4%) in 4 out of 6 projects compared to handcrafted fuzzers while containing interesting APIs for developers. More importantly, UTopia found 123 new bugs in 55 open source projects.

References

[1] Cve-2019-5063. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5063. Accessed: 2021-11-30.

[2] Oss-fuzz. https://github.com/google/oss-fuzz, . Accessed: 2019-02-03.

[3] Oss-fuzz integration rewards. https://google.github.io/oss-fuzz/getting-started/integration-rewards/, . Accessed: 2021-11-23.

[4] Tizen fuzzing dashboard. https://dashboard.tizen.org/fuzz.code, . Accessed: 2021-11-16.

[5] Tizen compliance tests. https://docs.tizen.org/platform/compliance/compliance-test, . Accessed: 2020-02-28.

[6] Boost test library. https://www.boost.org/doc/libs/1_54_0/libs/test/doc/html/index.html. Accessed: 2021-10-01.

[7] Google test. https://github.com/google/googletest.

[8] LAF-INTEL: Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/.

[9] Region: The connected subgraphs of a control flow graph that has exactly two connections to the remaining graph. https://llvm.org/doxygen/RegionInfo_8h_source.html. Accessed: 2021-12-01.

[10] The art, science, and engineering of fuzzing: A survey. 2018.

[11] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[12] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 975–985, 2019.

[13] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[14] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[15] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[16] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic Fuzzer Generation. In *Proceedings of the 29th USENIX Security Symposium (Security)*, pages 2271–2287, Boston, MA, Aug. 2020.

[17] J. Jang and H. K. Kim. Fuzzbuilder: automated building greybox fuzzing environment for c/c++ library. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 627–637, 2019.

[18] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[19] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[20] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[21] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1949–1966, 2019.

[22] V. J. Manès, S. Kim, and S. K. Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1024–1036, 2020.

[23] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[24] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 861–875, 2014.

[25] K. Serebryany. libFuzzer–a library for coverage-guided fuzz testing, 2015. https://llvm.org/docs/LibFuzzer.html.

[26] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. *NDSS*, 2020. doi: 10.14722/ndss.2020.24422.

[27] M. Zalewski. AFL: American Fuzzy Lop, 2014. http://lcamtuf.coredump.cx/afl/.

[28] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *Proceedings of the 30th USENIX Security Symposium (Security 2021)*, pages 2811–2828, Aug. 2021.

[29] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang. Intelligen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 318–327. IEEE, 2021.

Appendix

```
TEST(CodingPathSync, SearchForHbdLbdMismatch) {
- const int count_tests = 10;
+ const int count_tests = fi1;
  for (int i = 0; i < count_tests; ++i) {
    CompressedSource enc(i); // NB. forces count_tests to be mutated

-   Decoder dec_hbd(0);
-   Decoder dec_lbd(1);
+   Decoder dec_hbd(fi2);
+   Decoder dec_lbd(fi3);

    // NB. a static loop variable, 3, is not chosen for mutation
    for (int k = 0; k < 3; ++k) {
      const aom_codec_cx_pkt_t *frame = enc.ReadFrame();
      std::vector<int16_t> lbd_yuv = dec_lbd.decode(frame);
      std::vector<int16_t> hbd_yuv = dec_hbd.decode(frame);

-     ASSERT_EQ(lbd_yuv, hbd_yuv);
+     FUZZ_ASSERT_EQ(lbd_yuv, hbd_yuv); // NB. enable early
       termination
    }
  }
}

+ DEFINE_PROTO_FUZZER(const mutation::APIArgument &mutation) {
    /* NB. fi1 <- 0..9999 (loop variable).
           fi2-fi11 are randomly generated based on their types */
    /* NB. mutate the fields in 'aom_image_t' */
+   fi4_field23 = mutation.fi4().field23().c_str());
+   fi4.img_data_owner = mutation.fi4().field24();
+   fi4.self_allocd = mutation.fi4().field25();
+   fi4.img_data = fi4_field23;
+   ...
+ }

const aom_codec_cx_pkt_t *ReadFrame() {
  static const int kWidth = 128;
  static const int kHeight = 128;

  uint8_t buf[kWidth * kHeight * 3] = { 0 };

- const int period = rnd_.Rand8() % 32 + 1;
- const int phase = rnd_.Rand8() % period;
- const int val_a = rnd_.Rand8();
- const int val_b = rnd_.Rand8();
+ const int period = fi5 % 32 + 1;
+ const int phase = fi6 % period;
+ const int val_a = fi7;
+ const int val_b = fi8;

  for (int i = 0; i < (int)sizeof buf; ++i)
    buf[i] = (i + phase) % period < period / 2 ? val_a : val_b;

- width_ = rnd_.PseudoUniform(kWidth - 8) + 8;
- height_ = rnd_.PseudoUniform(kHeight - 8) + 8;
+ width_ = rnd_.PseudoUniform(fi9 - 8) + 8;
+ height_ = rnd_.PseudoUniform(fi10 - 8) + 8;

- aom_image_t img;
- aom_img_wrap(&img, format_, width_, height_, 0, buf);
- aom_codec_encode(&enc_, &img, frame_count_++, 1, 0);
+ aom_image_t img = fi4;
+ aom_img_wrap(&img, format_, width_, height_, fi9, buf);
+ aom_codec_encode(&enc_, &img, frame_count_++, fi10, fi11);

  aom_codec_iter_t iter = NULL;
  const aom_codec_cx_pkt_t *pkt = NULL;
  do {
    pkt = aom_codec_get_cx_data(&enc_, &iter);
  } while (pkt && pkt->kind != AOM_CODEC_CX_FRAME_PKT);
  return pkt;
}
```

Listing 1: A diff between the unit test of libaom and the generated fuzz driver of UTopia.

#	Library	Description	Status
1	assimp	Infinite loop	Confirmed
2-13	assimp	Access pointer without nullptr check	12 Reported
14-24	assimp	Free memory that is not located on the heap	11 Reported
25-44	assimp	Access pointer without nullptr check	20 Patched
45-52	assimp	Free memory that is not located on the heap	8 Patched
53-56	assimp	OOB write due to wrong type conversion of size	4 Patched
57	ledger	Divide by zero	Patched
58	libaom	OOB write due to wrong type conversion of size	Fixed
59	libaom	OOB write due to wrong type conversion of size	CVE-2021-30474
60	libaom	Access pointer without `nullptr` check	CVE-2021-30475
61	libaom	Access pointer without `nullptr` check	Fixed
62-66	libaom	Access pointer without `nullptr` check	5 Reported
67, 68	libaom	Divide by zero	2 Reported
69	libaom	Free memory that is not located on the heap	CVE-2021-30473
70	libaom	Memory access w/o boundary check	Fixed
71-74	libaom	Memory access w/o boundary check	4 Reported
75	libaom	Memory access w/o boundary check	Patched
76	libhtp	OOB write due to wrong type conversion of size	Fixed
77	libhtp	Memory access w/o boundary check	Fixed
78	libphonenumber	Access pointer without `nullptr` check	Fixed
79	libphonenumber	Memory access w/o boundary check	Fixed
80	libvpx	frees memory that is not located on the heap	Fixed (Requested CVE)
81	libvpx	Memory access w/o boundary check	Fixed (Requested CVE)
82	Node.js	Invalid format	Reported
83-85	openh264	OOB write due to wrong type conversion of size	3 Reported
86	openh264	Access pointer without `nullptr` check	Reported
87	openh264	Memory access w/o boundary check	Reported
88	openh264	Memory access w/o boundary check	Fixed
89	puffin	Memory access w/o boundary check	Reported
90	tesseract	OOB write due to wrong type conversion of size	Fixed
91	tesseract	Access pointer without `nullptr` check	Reported
92-94	tesseract	Memory access w/o boundary check	3 Reported
95	tesseract	Memory access w/o boundary check	Confirmed
96	tesseract	Memory access w/o boundary check	Fixed
97	tesseract	Memory access w/o boundary check	Patched
98	uriparser	Access pointer without `nullptr` check	CVE-2021-46141
99	uriparser	Access pointer without `nullptr` check	CVE-2021-46142
100, 101	vowpal-wabbit	Divide by zero	2 Fixed
102	vowpal-wabbit	Infinite loop	Reported
103	wabt	Infinite loop	Reported
104	wabt	Memory access w/o boundary check	Reported
Tizen			
105	libtbm	Access pointer without `nullptr` check	Fixed
106	yaca	Wrong type conversion of size	Fixed
107	privilege-info	Access pointer without `nullptr` check	Fixed
108	libmm-fileinfo	Memory access w/o boundary check	Fixed
109	alarm-manager	Access pointer without `nullptr` check	Fixed
110, 111	context	Missing exception handler	2 Fixed
112	context	No check on the return value causing free arbitrary memory address	Fixed
113	mediatool	Memory access w/o boundary check	Fixed
114	pkgmgr-info	Access pointer without `nullptr` check	Fixed
115, 116	system-settings	Access pointer without `nullptr` check	2 Fixed
117	mime-type	Missing exception handler	Fixed
118	sensor	Access pointer without `nullptr` check	Fixed
Rejected by maintainers			
R1	Node.js	Fell into an infinite loop. Directly tested internal APIs	Rejected
R2	leveldb	Fell into an infinite loop. Directly tested internal APIs	Rejected
R3	tesseract	Memory access w/o boundary check. Directly tested internal APIs	Rejected
R4, R5	libaom	Memory access w/o boundary check. Directly tested internal APIs	2 Rejected

Confirmed: The maintainer confirmed the bug but it has not been fixed yet.
Rejected: The maintainer didn't accept a report as a bug (e.g., API misuse).
Fixed: The maintainer accept a report as a bug and fix the reported issue.
Patched: The bug was also noticed and patched by maintainers before we reported the bug.

Table VII: Bugs in OSS discovered by UTOPIA.

Acronym	Description	# Defs
OOS	Assign statements in header files or out of project files	735
NP	Null pointer is assigned	297
EO	Assignment with return or output param of an external function (no input param)	219
FP	Type is a function pointer	168
CTC	Constants determined in compile time	44
RI	Values depend on ignored values (e.g. ArrayLen of ignored Array)	8

Table VIII: Definitions in excluded test cases in 25 projects.

Root cause	Detail	Effect	Cnt
Pointer access w/o validation	No `nullptr` check	SEGV	7
	No check on the returned error code	SEGV	1
Inappropriate type conversion	Array index conversion from big unsigned integer to signed negative integer	SEGV	1
Memory access w/o boundary check	size argument with memcpy	BoF	1
	struct field used as index of a fixed-sized array	BoF	1
Missing exception handler	for an error thrown	Abort	3

Table IX: Bugs in Tizen discovered by UTopia.

Cause	Detail	# Count (%)	Description
API semantic	Missed attributes	78 (33.3%)	Related to the accuracy of the static analyzer
	Internal object	71 (30.3%)	If parameter is stored in an internal object and referenced by the other API, UTopia does not cover
	New type of API attribute	30 (12.8%)	UTopia can be covered later Not implemented yet
	Nested loop	10 (4.3%)	The input value used as the loop count is too large UTopia can adjust it smaller the limit of value
Etc	Related UT code	29 (12.4%)	UT misses API call validation.
	Bug	8 (3.4%)	Suspected as a bug, but haven't reported yet
	No analysis	8 (3.4%)	Not enough information contained in the log to find the root cause
Total		234 (100%)	

Table X: Statistics of spurious crashes in 5 projects with most occured (libaom/assimp/openh264/tesseract/wabt)

A. A bug in a libnode API

A UTopia-generated fuzz driver for libnode (C++ implementation of Node.js) tested the `URL` constructor with two string arguments, `"/\nStrace: \n"` and `"file: //1h\333\207eam‘,ar"`, for input and base, respectively. The member `context` in class `URL` is initialized during parsing `base`, thereafter, a member `path` in `context` is accessed during the second parsing with `input`. At that instance, SEGV was triggered because `path` was set to `nullptr` during the first parsing.

Library	**OSS-Fuzz**		**UTopia**		
	#Drivers (APIs)	Coverage	#Drivers (APIs)	Coverage	Unique Coverage
assimp	1 (4)	6.5%	250 (97)	39.9%	33.8%
libphonenumber	1 (4)	28.9%	246 (143)	62.9%	37.7%
wabt	1 (1)	22.0%	80 (61)	25.5%	13.7%
leveldb	1 (15)	61.8%	175 (91)	85.3%	23.7%
libhtp	1 (27)	58.1%	336 (191)	78.0%	22.5%
uriparser	3 (17)	88.1%	82 (24)	92.4%	4.9%
muduo	1 (3)	4.9%	5 (13)	13.7%	9.8%

Table XI: Results of a single 24 hour fuzzing run of the remaining seven libraries with OSS-Fuzz drivers alongside their UTopia counterparts.

B. Generated drivers from FuzzGen PoC code

We had applied FuzzGen to the 40 UT of libvpx that UTopia transformed into fuzz drivers. However, as this process required several manual adjustments to fix various issues but did not provide any meaningful results, we moved on from pursuing this line. The details of the attempts are stated herein.

Manual fixes to enable FuzzGen for UT code analysis. For the FuzzGen PoC code to be capable of analyzing the UT code, we required to manually discover and disintegrate the header inclusion loops within the gtest framework headers as FuzzGen assumed no inclusion loops and was trapped in an infinite loop in case of encountering one (as it performed simple text parsing of the headers, #include guards do not prevent loops). This seemed a minor implementation detail and a straightforward fix, and thus, we applied the fix. Next, if we run FuzzGen after breaking the loops, it determined only 2 of the 40 UT with root functions worth analyzing. This is because FuzzGen rejected the functions within anonymous namespaces, as normally those would only have internal linkage and not be a root function because they would not be externally visible. However, a UT code employing UT frameworks may contain their test code within anonymous namespaces to avoid name collisions with other test code, while still being capable of exposing their code for the UT framework to run via registration macros. This appeared as an implementation detail that is specific to the UT framework, and thus, it did not seem fair to write FuzzGen off at this point. Therefore, to expose the test code to FuzzGen, we moved all test-related functions out of anonymous namespaces. Subsequently, FuzzGen could examine all 40 UT codes.

Generated fuzz drivers. Among the 40 UT code, FuzzGen could extract API sequences from 26, However, it failed to locate library API calls in 14 of them primarily because of the cross-source file issue discussed earlier (D1 in §II-A). If we generated fuzz drivers from each of the 26, only 6 could be run with straightforward fixes guided by compile errors which we determined could be considered as minor implementation issues due to its PoC nature (missing header inclusions, incorrect type casts, incorrect dereferences, etc.) Among the remaining 20, 13 required compile error fixes depending on conscious decisions (decisions on the procedure of fixing missing call parameters, resolving function pointers to functions within the consumer code, fixing incorrect struct member analysis, etc.), which we could not fix without affecting the capability of the resulting fuzz driver. In contrast, the other 7 of the 20 could be eliminated of compile errors but they suffered from severe spurious crashes (owing to incomplete initialization or invalid API sequences), which could not be fixed fairly without affecting their capabilities. These errors mostly stemmed from the implementation decision of FuzzGen in not considering the consumer (in this case UT) that defined the compound types into account during its consumer data flow analysis, which breaks the related API call parameters in generated drivers. Among the 6 executable fuzz drivers, unfortunately, none were operation worthy as none of them had been generated to receive a fuzzing input. This is because the analysis of FuzzGen determined that the acquired APIs did not pose arguments that could accept random values. This was because FuzzGen could not observe any operations performed in the def-use chains of the arguments within the library caused by its incapability to resolve indirect calls within the APIs, where the necessary operation statements indicating the arguments may be fuzzed. Among the 26 generated drivers, only two drivers were generated to accept fuzzing input but unfortunately, fixing them required decisions, as discussed earlier, seemed beyond the simple implementation. Thus resolving them would invalidate our goal of fair comparison, because the resulting drivers could manifest drastically different capabilities, depending on the method of resolution.

Coalescing UT-based drivers. As the individual transformation of consumer code into fuzz driver is not the fundamental concept driving FuzzGen, we tried coalescing the UT-based drivers and examined its coalescing of the API sequence. In case of assuming all the errors to be fixed, the above-mentioned procedure will succeed. However, after examination, the results revealed that it would still yield severe spurious crash because of invalid API usage. This was because during the coalescence of the UT API sequences, FuzzGen coalesced two `vpx_codec_decode()` calls where one of the calls was from a loop of `vpx_codec_dec_init_ver()`, `vpx_codec_decode()`, and `vpx_codec_destroy()` in order. As the call graph (CG A) for this loop contains a backward edge from `vpx_codec_destroy()` to `vpx_codec_dec_init_ver()`, FuzzGen has rotated the graph after coalescing the `vpx_codec_decode()` call and ends up with a sub-sequence in the order of `vpx_codec_decode()`, `vpx_codec_destroy()`, and `vpx_codec_dec_init_ver()`, which caused a spurious crash because of attempting to dereference a member of an uninitialized structure. In the entire coalesced graph, this subsequence followed a `vpx_codec_dec_init_ver()` call from the other call graph (CG B) of the coalesced `vpx_codec_decode()` calls but the structure that is initialized by this initialization call is used only by the subsequence of API calls from its own original call graph (CG B) and not by the calls in the rotated sequence (CG A), which does not help with the spurious crash. This issue is acknowledged in the study proposing FuzzGen and a manual fix can probably correct the coalesced sequence. However, this would be detrimental to the scalability, if we considered this approach for automatic fuzz driver generation.