DAISY: Effective Fuzz Driver Synthesis with Object Usage Sequence Analysis

Mingrui Zhang[1], Chijin Zhou[1], Jianzhong Liu[1], Mingzhe Wang[1,2], Jie Liang[1,2], Juan Zhu✉ [3] and Yu Jiang[1]

[1]BNRist, School of Software, Tsinghua University
[2]ShuiMuYuLin Co. Ltd, Beijing, China
[3]Hubei University of Arts and Science, China

***Abstract*— Fuzzing is increasingly used in industrial settings for vulnerability detection due to its scalability and effectiveness. Libraries require driver programs to feed the fuzzer-generated inputs into library-provided interfaces. Writing such drivers manually is tedious and error-prone, thus greatly hindering the widespread use of fuzzing in practical situations. Previous attempts at automatic driver synthesis perform *static* analysis on the libraries and their consumers. However, a lack of *dynamic* object usage information renders them ineffective at generating interface function calls with correct parameters and meaningful sequences. This severely limits fuzzing's bug-finding capabilities and can produce faulty drivers.**

In this paper, we propose DAISY, a driver synthesis framework, which extracts dynamic *object usage sequences* of library consumers to synthesize significantly more effective drivers. DAISY uses the following two steps to synthesize a fuzz driver for a library. First, it models each object's behaviors into an object usage sequence during the execution of its consumers. Next, it merges all the extracted sequences and constructs a series of interface calls with valid object usages based on the merged sequence. We implemented DAISY and evaluated its effectiveness on real-world libraries selected from both the Android Open Source Project (AOSP) and Google's FuzzBench. DAISY's synthesized drivers significantly outperform drivers produced by other state-of-the-art fuzz driver synthesizers. In addition, on applying DAISY to the latest versions of those extensively-fuzzed real-world libraries of the benchmark, e.g. *libaom* and *freetype2*, we also found 9 previously-unknown bugs with 3 CVEs assigned.

I. INTRODUCTION

Fuzzing is a popular and efficient technique widely used in vulnerability detection. Thousands of bugs in diverse areas have already been discovered by various fuzzers [1], [2], [3], [4], [5], [6]. Fuzzers test programs by feeding programs or libraries a large number of generated inputs to trigger bugs. As it is a dynamic testing technique, testers need to have an executable program to feed fuzzer-generated inputs. For libraries that do not have executable programs available, testers need to construct a fuzz driver to act as an entry function that feeds fuzzer-generated inputs into interfaces of target libraries.

Although fuzzing has made significant progress, its effectiveness relies heavily on the quality and diversity of fuzz drivers. Unfortunately, writing fuzz drivers is often tedious and labor-consuming while its quality is highly dependent on the writer's prior knowledge and experience with the relevant libraries. Writing high-quality drivers requires a thorough understanding of the library's APIs to organize them into an effective fuzz driver. Usually, high-quality fuzz drivers are lengthy and functionally complex. For example, the official fuzz driver of *freetype2* takes more than 400 lines of code to build the context of its core interface. These issues currently hinder fuzzing's usability and effectiveness. Therefore, writing or generating a high-quality fuzz driver is a hot topic.

There have been a few attempts at generating fuzz drivers automatically. These methods can be categorized into two types based on how they extract potential interface information, i.e. *library-based* synthesis such as FUDGE [7] and IntelliGen [8], and *consumer-based* synthesis such as FuzzGen [9]. The former approach extracts interesting interface calls to the target library and extracts their relevant snippets to synthesize drivers, but experiences difficulties in producing effective call sequences due to insufficient runtime information. In contrast, the latter approach only performs *static* analysis on the consumer's code base, and thus may fail to capture *dynamic* usage information of all objects. Specifically, static-based synthesis methods experience difficulties as they are less likely to determine valid calling chains that contain a usage pattern of certain objects, partially due to indirect function calls and ambiguous type information. For instance, inter-procedural data flows would be ignored by static synthesizers in the following example. Listing 1 presents a unit test segment from *libxml2*, an XML parser library widely used in industrial systems. *reader_1* and *reader_2* are two reader objects of the same type. They are passed to interface functions of *libxml2* in a certain order. To synthesize a driver, static consumer-based approaches abstract the argument dependencies based on their types and use the same interface sequence as this unit test. However, in this example, these two variables with the same type can *fool* the synthesizer into generating a buggy driver. Specifically, the static synthesizer observes that the reader object can be used in the order of *create* $\rightarrow$ *use* $\rightarrow$ *free* $\rightarrow$ *use*. Therefore the driver contains a use-after-free bug and is thus faulty.

To overcome this issue, we need to extract object usage sequences dynamically and construct drivers containing a valid sequence, where we encounter the following challenges:

- **Effective object usage information collection**. Conven-

*Juan Zhu is the corresponding author. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730).

```
/* interfaces exported by the library itself*/
struct xmlTextReaderPtr {};
xmlTextReaderPtr xmlNewTextReaderFilename(
  const char*);
int xmlTextReaderRead(xmlTextReaderPtr);
void xmlTextReaderNodeType(xmlTextReaderPtr);
void xmlFreeTextReader(xmlTextReaderPtr);

static void handleFile(
  const char *filename_1, const char *filename_2
) {
  xmlTextReaderPtr reader_1 =
    xmlNewTextReaderFilename(filename_1);
  xmlTextReaderPtr reader_2 =
    xmlNewTextReaderFilename(filename_2);
  int ret = xmlTextReaderRead(reader_1);
  // ... ...
  xmlTextReaderNodeType(reader_1);
  xmlFreeTextReader(reader_1);
  int ret = xmlTextReaderRead(reader_2);
  // ... ...
  xmlFreeTextReader(reader_2);
  // ... ...
}
```

Listing 1 Code segment of a unit test from *libxml2*

tional static flow analysis cannot obtain implicit or interprocedural object usage and dependency information reliably. However, switching to dynamic analysis directly raises concerns regarding its precision and overhead. Therefore, performing high-precision dynamic object usage information collection in a cost-efficient way is critical to the practicality of our approach.

- **Driver synthesis with actual usage sequences**. Synthesized drivers should contain object usages reflecting actual data usage information extracted from the library consumer. However, the library consumer generally does not contain any entries for fuzzer's input. It is necessary to propose a novel synthesis solution so that the synthesized driver not only follows the object usage sequence from real-world consumers but also properly bridges the gap between the fuzzer's input and the driver's entry.

In this paper, we propose DAISY, a novel driver synthesis framework for C/C++ projects, to make automated driver synthesis practical and effective in industrial scenarios. Our key insight is to model the sequence of each object's behaviors, including construction, utilization, and destruction (named **Object Usage Sequence, *OUS***), during the execution of library consumers. Unlike static methods that conventional synthesizers use, such an *object usage sequence* can preserve the original valid usage sequence and distinguish different objects even if they have the same type. To achieve this goal, DAISY first performs instrumentation on the consumers of the target library and constructs the *OUS* during execution. Then, DAISY merges all *OUS*es into one and synthesizes a fuzz driver based on the sequence model. Additionally, a runtime library is supported to help DAISY intercept object constructions in the driver so that the fuzzer's input can be dispatched into these objects. We implement DAISY based on the LLVM compiler framework. DAISY is made up of two LLVM passes and a runtime support library. One pass aims to perform instrumentation on the library consumer to model the *OUS*. The other pass is used to merge multiple *OUS*es into one and synthesize a fuzz driver according to the sequence model. The runtime library is designed to determine how to assign values for all objects allocated by the fuzz driver based on the fuzzer's input.

We evaluate DAISY's effectiveness on real-world programs from Google's FuzzBench and Android Open Source Project (AOSP). On average, DAISY covered 1.39-2.93 times more basic blocks over other state-of-the-art fuzz driver synthesizers, i.e., FuzzGen and IntelliGen, and detected 9 previous-unknown bugs with 3 CVEs assigned. In summary, we make the following contributions:

- We propose a fuzz driver generation framework DAISY which dynamically constructs the *OUS* from the library consumers and synthesizes fuzz drivers according to the acquired information.
- We implement DAISY on the top of the LLVM compiler infrastructure, and open-source the relevant artifacts on github [10]. DAISY's implementation consists of sequence extraction and driver synthesis modules.
- We evaluate DAISY on FuzzBench and AOSP against state-of-the-art driver synthesizers. DAISY detected 9 previous-unknown bugs with 3 CVEs assigned. The results show that DAISY is able to synthesize practical and effective fuzz drivers in industrial scenarios.

The rest of the paper is organized as follows: Section II introduces the background and related work. Section III describes the motivation behind DAISY. Section IV outlines DAISY's design, including object usage sequence modeling and driver synthesis. Section V evaluates DAISY on AOSP and FuzzBench and compares its effectiveness against FuzzGen, IntelliGen, and expert-written drivers. Section VI discusses the problems we met and the lessons we learned.

II. RELATED WORK

Fuzzing in real-world practice. Fuzzing is an automated program vulnerability detection technique. Many fuzzers have been developed with varying degrees of success, including AFL [11], Honggfuzz [12], libFuzzer [13], etc. Efforts to combine the power of multiple fuzzers have achieved substantial improvement over using a single fuzzer. For instance, Google's OSS-Fuzz [1] combines AFL and libFuzzer to fuzz open-source programs and has discovered thousands of bugs over a period of around 5 years. Fuzzers generally aim to explore as many program states as possible, since a bug cannot be triggered unless its location can be visited through an execution path. As the number of program states can be infinite, researchers and developers have devised a number of techniques [14], [15], [16], [17], [18], [19], [20] that allow fuzzers to cover more code and trigger more bugs. Due to its scalability, fuzzing has been successfully applied to many critical areas [21], [22], [23], [24]. To fuzz a library, testers write a fuzz driver to pass the fuzzy input to the library. Hence, a fuzzer's effectiveness at testing libraries greatly depends on the driver programs' quality.

Fuzz driver synthesis frameworks. Automatically synthesizing a high-quality fuzz driver is a hot topic in the fuzzing domain, and there are a few attempts at automated fuzz driver synthesis. These synthesizers can be divided into two categories, i.e. library-based driver synthesis and consumer-based driver synthesis. FUDGE [7] and IntelliGen [8] are two library-based driver synthesis frameworks. FUDGE scans the source code of the library, identifies interface calls containing arguments that take data buffers, and extracts the relevant code segments to synthesize a fuzz driver. IntelliGen directly calls an API function based on a heuristic ranking and assigns values to the arguments using a lazy-store technique. In contrast, FuzzGen [9] statically builds a dependence graph by analyzing library consumers and uses this information to construct interface invocations in the synthesized fuzz drivers. By utilizing static consumer information, FuzzGen improves upon FUDGE and allows fuzzers to test more code in the target libraries. APICraft [25] and GraphFuzz [26] are the other two fuzz driver synthesis frameworks. The former primarily aims to generate fuzz drivers for closed-source SDKs on Apple's MacOS platform, and the latter produces a fuzz driver by constructing the schema of the target library and mutating the data flow contained by the schema. Unlike these static approaches, DAISY focuses on modeling each object's behaviors during the execution of library consumers, and further facilitating fuzz driver synthesis.

Unit test generation tools. These tools automatically generate interface invocations with predefined static values in the form of unit tests. For instance, Randoop [27] generates high-quality unit tests by merging previous-generated inputs, Testful [28] provides test cases for stateful systems by reusing the same states, Evosuite [29] hybrids different test cases to achieve high code coverage, and Tautoko [30] dynamically mines type-state models from existing valid sequences of method calls to generate unit tests. The main difference between DAISY and the unit test generation works are the scopes and goals. In terms of scopes, the fuzz drivers generated by DAISY allow fuzzers to explore as much of the program's state as possible to discover vulnerabilities, while unit tests generated by these frameworks allow developers to detect functional bugs by comparing library behaviors with manually defined oracles. In terms of goals, DAISY generates invocation sequences based on off-the-shelf library consumers, while unit test generation works generate invocation sequences based on try-and-error evolution.

III. MOTIVATION

Current driver synthesizers construct fuzz driver programs based on interface signatures or library consumers. However, their usability is limited in real-world scenarios. Using the example presented in the introduction, Listing 1 is a segment from a unit test in *libxml2*, an XML parser library widely used in industrial settings. The function declarations in Lines 2-8 are part of the interface signatures exported by *libxml2*, while the function in Lines 11-27 is a unit test that invokes some interfaces to test their functionality. *xmlNewTextReaderFilename()* opens a file and returns a pointer to a reader object, *xmlTextReaderRead()* and *xmlTextReaderNodeType()* are two interfaces that take a pointer to a reader object as argument and from which read an XML object, *xmlFreeTextReader()* takes a pointer to a reader object as argument and frees all the corresponding object values.

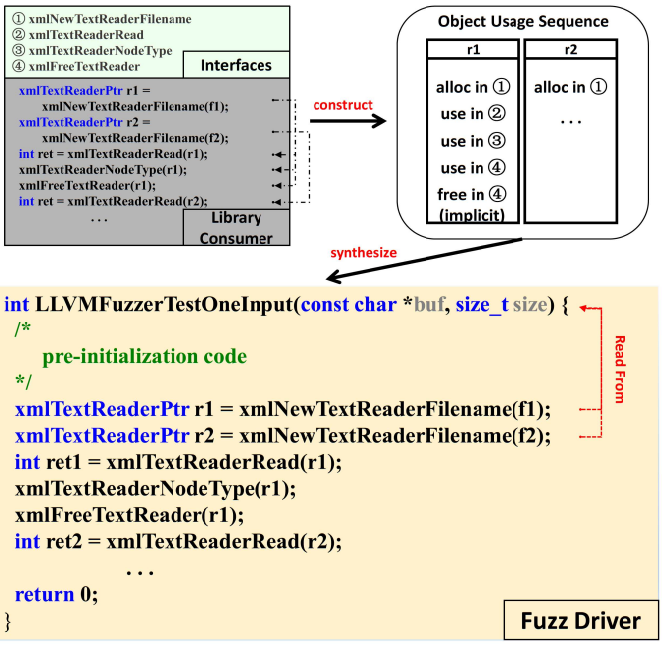

Fig. 1: A demonstration of object usage sequence for the motivating example. To synthesize a quality fuzz driver, DAISY first records all objects' usage based on the library consumer and then follows the usage sequence to build a driver skeleton.

Although there are only a few exported interfaces and the unit test is quite straightforward, existing synthesizers fail to synthesize an effective fuzz driver. For instance, FUDGE scans all the library's interfaces and extracts the interfaces which have the argument list *(const char*, size_t)* as the entry functions. Unfortunately, there are no such interface functions in this library, thus FUDGE is incapable of synthesizing any drivers. IntelliGen synthesizes a driver in which a series of variables are initialized and then passed to a single library interface (e.g. *xmlTextReaderRead()*) as arguments. However, these variables are initialized as random values generated by fuzzers, and thus can hardly bypass shallow sanity checks due to the lack of valid initialization. In this example, *xmlTextReaderRead()* would immediately return an error code after the input violates preliminary sanity checks. FuzzGen extracts a typed-based Abstract API Dependence Graph from the library's unit tests and synthesizes a sequence of interface invocations. However, fuzzing a driver synthesized by FuzzGen will result in many false positives since it only focuses on type-level dependence relations instead of object-level usage sequences. In this example, FuzzGen generates a sequence similar to the following invocation sequence:

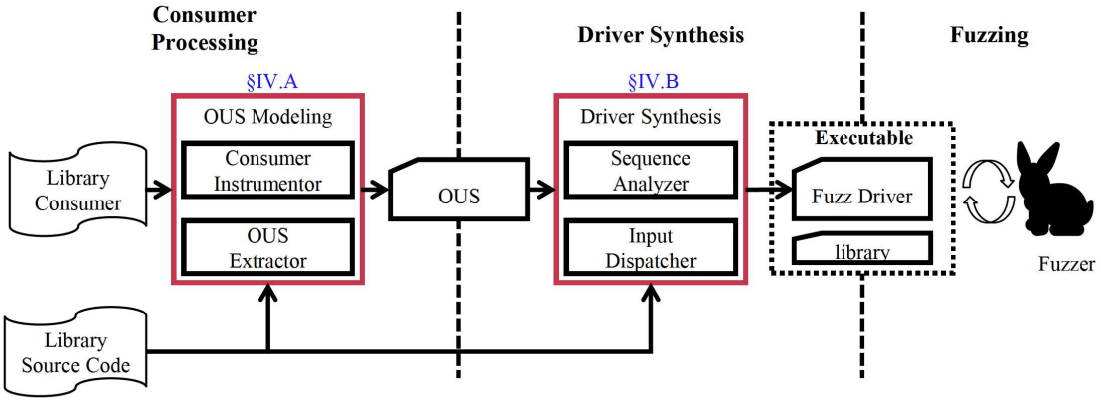

Fig. 2: The overall architecture design of DAISY, which operates in two stages. First, it produces the *Object Usage Sequence* by performing instrumentation and executing library consumers to extract relevant usage information. Then, it leverages the sequences to synthesize a fuzz driver. The Input Dispatcher transfers the inputs generated by the fuzzer to arguments used by library interface functions at runtime.

```
var = xmlNewTextReaderFilename(...);
xmlFreeTextReader(var);
xmlTextReaderRead(var);
```

which means that an *xmlTextReaderPtr* object is created, destroyed, and then used again, resulting in a use-after-free bug.] Such a driver will lead to false positives and reduce a fuzzer's effectiveness.

According to our observation, the root cause of the ineffective synthesis methods is that they do not take into account *Object Usage Sequences*. Previous work generally generates drivers that break existing object usage patterns to fill the contents of selected variables with the fuzzer's input. Such an approach can conveniently overcome the differences between the fuzzer's input and the driver's input variables. However, library consumers (i.e. unit tests in this example) often follow a semantic pattern to construct and use objects as well as invoke interfaces. Breaking this pattern will lower the effectiveness of synthesized fuzzers and greatly impact fuzzing's performance. We use the term *Object Usage Sequence* to refer to the sequence of each object's behaviors, including construction, utilization, and destruction.

In this paper, we propose DAISY to make fuzz driver synthesis practical in industrial scenarios. Fig. 1 demonstrates how it operates on the motivating example. First, it executes the library consumer and constructs *object usage sequences* for each object. For each object, DAISY tracks its interface-level data usage. Take *r1* as an example, it is first created in interface ①, then used in interfaces ②, ③, and ④ successively, and finally implicitly freed in interface ④. We illustrate DAISY's detailed design of object usage sequence modeling in Section IV-A. Next, DAISY synthesizes a fuzz driver following the sequence model. To bridge the gap between the fuzzer's input and the driver's used values, all the sources of object allocation are intercepted by DAISY's runtime library. In these interceptors, DAISY assigns a part of the input's buffer to the object allocation. The detailed design is illustrated in Section IV-B.

IV. DAISY DESIGN

In this section, we describe the design of DAISY in depth. Fig. 2 shows the overall architecture design of DAISY. DAISY synthesizes fuzz drivers in a two-step procedure. (1) In the first step, the *Consumer Instrumentor* performs instrumentation on the consumer programs at every construction, destruction, and utilization position. Then, the *OUS Extractor* dynamically extracts the *OUS*es during the execution of the library consumer programs. (2) In the second step, the *Sequence Analyzer* merges all *OUS*es extracted by the *OUS Extractor* into a single *OUS*, and produces a fuzz driver skeleton using the information recovered from the merged *OUS*. Then, the *Input Dispatcher* intercepts all objects' construction and fills the relevant values with contents taken from the input's buffer to bridge the gap between the fuzzer's inputs and the driver's utilized objects. After the two-step procedure, DAISY produces a fuzz driver that can be used as a fuzzing target.

A. Object Usage Sequence Modeling

During *OUS* modeling, DAISY first performs instrumentation on the target library's consumers through the *Consumer Instrumentor* component. Then DAISY dynamically extracts *object usage sequences* through the *OUS extractor* component.

Consumer Instrumentor. The *Consumer Instrumentor* performs instrumentation of each object's construction, destruction, and utilization behavior in the library consumers. Specifically, a command *cmd* will be instrumented if it matches one of the following patterns:

- Construction patterns.
  - *cmd* defines a local variable, which allocates a memory block on the stack.
  - *cmd* calls *malloc()* to get a pointer pointing to a memory chunk on the heap.
  - *cmd* calls *fopen()* to obtain a file object.
  - *cmd* dereferences a pointer, which points to a non-recorded file object or memory chunk.

- Destruction patterns.
  - *cmd* returns from a function, which automatically deallocates all local variables defined in this function.
  - *cmd* calls *free()* to free a buffer chunk on the heap.
  - *cmd* calls *fclose()* to close a file.
- Utilization patterns.
  - *cmd* stores a value to a tracked object.
  - *cmd* calls a function defined in the target library.

Note that DAISY only tracks interface functions defined in the target library's archive file (.a) or shared object file (.so), so functions from external libraries, e.g. *printf()*, will be omitted.

***OUS* Extractor**. After performing instrumentation on consumers, DAISY executes them and uses the *OUS Extractor* to extract *OUS*es. Algorithm 1 shows the details of how the *OUS Extractor* works during a consumer's execution.

Algorithm 1: Object Usage Sequence Extractor

```
   Input: The consumer's execution trace trace
   Output: The object usage sequence OUS extracted
           from the consumer
1  OUS = emptyList();
2  pool = emptySet();
3  foreach command cmd in trace do
4      if cmd matches construction patterns then
5          obj = cmd.getObject();
6          pool.add(obj);
7          OUS.add(obj);
8      else if cmd matches destruction patterns then
9          obj = cmd.getObject();
10         pool.remove(obj);
11     else if cmd matches utilization patterns then
12         OUS.addUtilization(cmd, pool);
13     end
14 end
15 return OUS;
```

The algorithm takes a consumer program's execution trace *trace* as the input and returns the extracted object usage sequence *OUS*. In line 3, *trace* is all LLVM-IR instructions executed by the consumer, and *cmd* is one instruction in *trace*. DAISY maintains an object pool *pool*, which assigns each object to a unique ID on its construction in line 6. When a new object is constructed, DAISY adds it to *pool* and dumps it into *OUS* as shown in lines 5-7. DAISY removes an object from *pool* when it is freed as provided in lines 8-10, but does not record the object-destruction operation in *OUS*. All objects will be freed at the end of the fuzz driver, as shown in Section IV-B. When the consumer utilizes an object, DAISY tracks the utilization command and dumps it into *OUS* in lines 11-12. Two kinds of utilization are tracked, i.e. (1) store operations and (2) function invocations. On detecting a store operation that stores a pointer value to any address, if the target address points to a tracked object, then DAISY appends the storage operation to *OUS*. When meeting a function invocation operation, if it calls a function defined in the target library, DAISY then appends the invocation operation to *OUS*. After the consumer program's execution, *OUS* is extracted from the target consumer and emitted for the next step.

B. Driver Synthesis

After extracting *OUS*es as shown in Section IV-A, DAISY's *Sequence Analyzer* reduces and combines the *OUS*es, then constructs a driver skeleton according to the relevant information. Next, DAISY's *Input Dispatcher* assigns variables' values extracted from the fuzzer's inputs.

Sequence Analyzer. Since a library may contain multiple consumer programs, the *Sequence Analyzer* first reduces every origin *OUS* and merges them, then constructs a skeleton driver program according to the merged *OUS*.

Reducing an OUS. The consumer programs of the target library may contain code irrelevant to driver synthesis, such as code performing command line input parsing or logging, thus their corresponding object usage information should be removed from the *OUS*. To remove this irrelevant information, DAISY first scans the raw *OUS*, marks all function calls as *useful*, and puts them into a FIFO queue. Then, DAISY picks an entry from the queue, marks any other unmarked entry which refers to at least one object of the selected entry as *useful*, and puts them into the same queue. DAISY repeats this progress until the queue is empty, thus any entry which is not marked as *useful* should be removed from the original *OUS*. After reducing the original sequence, DAISY joins all nodes that represent the same function call together. For instance, assume that the entries *A* and *B* call the same function with the same arguments, then DAISY replaces all edges from or to *B* with edges from or to *A*, and erases *B* from the sequence. This process also makes all function call records unique, facilitating convenient merging with another *OUS*.

Merging multiple OUSes. Instead of synthesizing many fuzz drivers from individual *OUS*es, DAISY merges all the *OUS*es and generates one fuzz driver from the merged *OUS*. DAISY merges two *OUS*es by grafting the same function call records, where the detail is shown in Algorithm 2.

Specifically, DAISY uses a greedy algorithm to find compatible function call records between two *OUS*es. DAISY first provides a function *callMatch()* to determine if two function call records are compatible. Two function call records can be compatible only when they have the same function callee, so *callMatch* returns FALSE immediately if the two function call records invoke different functions, as shown in line 2-3. In addition, two matched function call records must also have bijective arguments. Therefore, for each corresponding argument object pair (*obj1*, *obj2*) in the two function call records, if an object does not match the other one, then *callMatch()* returns FALSE, as shown in lines 6-8. The *match* in line 6 refers to if two objects *obj1* and *obj2* can be represented by one object in the merged *OUS*. To determine this, DAISY logs all object pairs ever matched. If they ever match each other, then they always will. Else if one of *obj1* and *obj2* ever matches another object *obj3*, then *obj1* and *obj2* are not matched. Otherwise, DAISY marks that *obj1* and *obj2* may

Algorithm 2: Merging two *OUS*es.

Input: Two object usage sequences *OUS1* and *OUS2*.
Output: The merged sequence

```
1  Function callMatch(call1, call2):
2      if call1.func != call2.func then
3          return FALSE;
4      end
5      foreach (obj1, obj2) in call1.args · call2.args do
6          if obj1 does NOT match obj2 then
7              return FALSE;
8          end
9      end
10     updateMatchingStates();
11     return TRUE;
12 end
13 foreach calls (call1, call2) in OUS1 × OUS2 do
14     if callMatch(call1, call2) then
15         merge(call1, call2);
16     end
17 end
```

match each other. If there is no contradiction after judging all object pairs in line 9, then DAISY logs all matched object pairs in line 10, and function *callMatch* returns TRUE in line 11. Else, function *callMatch* returns FALSE in line 7. For each call record pair (*rec1*, *rec2*) in a full permutation of two sequences (*OUS1*, *OUS2*), if *callMatch(rec1, rec2)* returns TRUE (line 14), which means *rec1* and *rec2* are compatible, DAISY merges the two function calls to connect the two *OUS*es in line 15.

Driver Synthesis. Once getting the merged *OUS*, DAISY constructs a fuzz driver's skeleton using the information retrieved from it. The details are shown in Algorithm 3. The *Sequence Analyzer* maintains an object pool to maintain all constructed objects. When retrieving an object from the *OUS*, DAISY adds an instruction to allocate a buffer or open a file according to the object's type, then adds the object into the pool as shown in lines 4-7. The utilization operations can be divided into two types: (1) store operations and (2) function invocations, as mentioned in Section IV-A. For each utilization operation, DAISY accesses its operands and processes them based on their types. If an operand is a pointer, then DAISY will retrieve the underlying object and fill the operand position with it. If an operand is a built-in type (e.g. *int* or *float*), then the operand position will be dispatched with a chunk of the fuzzer's generated input. Finally, DAISY destructs all objects to avoid memory leaks, as shown in lines 13-15.

Input Dispatcher. The *Sequence Analyzer* generates a fuzz driver skeleton ready to run. However, to execute the fuzz driver, we also need to assign values for all objects at runtime. The *Input Dispatcher* is designed to assign values for objects based on the generated input. This process should be both random and stable. On the one hand, the dispatcher should accept the fuzzer's input, in which the value is randomized. On the other hand, the dispatch strategy should be stable for bug reproduction. The *Input Dispatcher* first records the buffer provided by the fuzz engine at the beginning of a fuzzing round. When a memory chunk is constructed, the *Input Dispatcher* reads the same number of bytes as the chunk's size and copies them to the chunk. When a new file is opened, the *Input Dispatcher* reads a 2-byte number as the size of the file, then reads the same number of bytes and saves them to the file. The input buffer provided by the fuzz engine may be not enough to provide all the data, hence the *Input Dispatcher* loops back to the front of the buffer upon reaching its end.

Algorithm 3: Synthesizing a Fuzz Driver.

Input: The merged object usage sequence *OUS*.
Output: The fuzz driver *driver*

```
1  pool = emptySet();
2  driver = emptyDriver();
3  foreach record rec in OUS do
4      if rec is construction then
5          obj = rec.retrieveObject();
6          driver.addConstrutInstr(obj);
7          pool.add(obj);
8      else if rec is utilization then
9          util = rec.retrieveUtilization();
10         driver.addUtilizeInstr(util);
11     end
12 end
13 foreach object obj in pool do
14     driver.addDestructInstr(obj);
15 end
16 pool.clear();
17 return driver;
```

C. DAISY *Implementation*

We implement DAISY using the LLVM compiler infrastructure will open-source it. As shown in Section IV, DAISY contains two main modules. One is the *OUS Constructor*, the other is the *Driver Synthesizer*, both of them are implemented as LLVM passes. Though most C++ programs call destructors implicitly in the source code, they are explicitly called in the LLVM-IR by a call-instruction. Hence DAISY can capture them on extracting *OUS*. When synthesizing fuzz drivers, DAISY calls the destructors explicitly by their mangled name. DAISY emits the driver as C++ code, benefiting testers in verifying and modifying the generated driver program manually.

During a single fuzzing round, DAISY maintains a hash set to record all allocated objects. It records any memory chunk on construction and removes it on destruction. After finishing a round of fuzzing, DAISY frees all allocated memory chunks in the hash set and closes all opened file descriptors.

V. EVALUATION

In this section, we evaluate the effectiveness of DAISY. We compare DAISY with existing fuzz driver synthesizers based on a variety of metrics to demonstrate the DAISY's

effectiveness. Moreover, we present several case studies to give a concrete demonstration of how it allows fuzzers to find unknown bugs.

A. Evaluation Setup

Experiment environment. We evaluate DAISY on a 64-bit machine with 128 cores (AMD EPYC 7742) and 500GiB of RAM running Linux 5.4.128 for the following experiments.

Target libraries. We conduct experiments on two well-known datasets for fuzzing: the Android Open Source Project (AOSP) and Google's FuzzBench [31]. Regarding FuzzBench, we select every program included in both Google's *fuzzer-test-suite* and FuzzBench as they have been carefully picked for evaluating fuzzing performance. All libraries are widely used in industry settings and have been extensively tested using OSS-Fuzz[1]. We use unit test cases, system test cases, and real-world applications from the code base of each library as its consumers.

Evaluation metrics. We use basic block coverage to evaluate the effectiveness of synthesis techniques. The coverage is collected by *llvm-cov* [32], a common measurement tool for evaluating fuzzing performance. We follow the fuzzing evaluation guidelines outlined by Klees et al. [33] to conduct each experiment over a period of 24 hours and repeat five times, which is an experiment setting widely used in fuzzing performance evaluation [31], [34], [35], [36], [37], [21], [38].

B. Overall Performance

TABLE I: Average basic block coverage on AOSP projects in a 24-hour fuzzing campaign and repeating five times using fuzz drivers generated by different techniques and written by expert. Entries with "✓" represent that the p-value of coverage between DAISY's driver and the corresponding driver is less than 0.05. "-" entries indicate that the synthesizer cannot produce a valid driver program for the project. Entries marked with "*" represent the result from FuzzGen's paper.

project	DAISY	FuzzGen	IntelliGen	Expert-Written
libaom	10771.2	4538*	-	19741.4
libavc	609	4097*	775.4 (✓)	507.4 (✓)
libgsm	76	557*	1594.4 (✓)	391.2 (✓)
libhevc	10122	6037*	580.6 (✓)	14584.4
libmpeg2	2685.6	958*	381.4 (✓)	2790
libpng	643.8	-	-	3455.4 (✓)
libopus	5505.8	2837*	95.2 (✓)	5439.4 (✓)
libvpx	1064.6	1840*	3269.6 (✓)	3299.4 (✓)
libxaac	13532	-	95.4 (✓)	12002.6
libxml2	5741.6	-	6108.4	11954.2 (✓)
Improvement	-	1.39	2.93	-0.32

We compare DAISY against two state-of-the-art fuzz driver synthesizers, FuzzGen and IntelliGen, on 10 external libraries selected from AOSP and 12 programs selected from FuzzBench. The overall results are presented in Table I and Table II. Due to the limitations of FuzzGen and IntelliGen, some programs are incompatible and cannot be successfully compiled. From the two tables, we can observe that:

TABLE II: Average basic block coverage on FuzzBench projects in a 24-hour fuzzing campaign and repeating five times using fuzz drivers generated by different techniques and written by expert. Entries with "✓" represent that the p-value of coverage between DAISY's driver and the corresponding driver is less than 0.05. "-" entries indicate that the synthesizer cannot produce a valid driver program for the project.

project	DAISY	IntelliGen	Expert-Written
boringssl	1012.8	-	2345.4 (✓)
freetype2	5407.6	-	12739.4 (✓)
harfbuzz	4400.8	4419.6	4595.8 (✓)
json	811.4	794 (✓)	903.8 (✓)
lcms	1573.2	376 (✓)	2811.2 (✓)
libarchive	2554	-	7704.8 (✓)
libjpeg	613.4	1215.8 (✓)	1092.4 (✓)
proj4	1382.4	76.4 (✓)	6909.2 (✓)
re2	3373.8	2381.8 (✓)	3501 (✓)
sqlite	3242.8	-	3141.4 (✓)
vorbis	146.4	108.2 (✓)	177.6
woff2	1835.6	-	1498
Improvement	-	1.81	-0.44

Compared with FuzzGen, DAISY can generate driver programs for a wider variety of libraries and achieves better block coverage. Though we have thoroughly tried to run FuzzGen on all tested projects, FuzzGen is ultimately incompatible with all the projects of FuzzBench. This is because FuzzGen requires a full-system sweep of library consumer information, for instance, the entire AOSP codebase for Android applications, to construct its dependence graph. However, the projects in FuzzBench depend on *glibc*, and thus extracting their dependence graphs is significantly more difficult. Therefore, we present the original data from the FuzzGen paper and calculate the corresponding basic block coverage in Table I. In contrast, DAISY is much easier to adapt to different libraries, as it only needs to conduct object usage analysis on the consumer programs rather than a full-system dependency analysis. As a result, DAISY can generate drivers for a large number of libraries in both AOSP and FuzzBench, where their performance reaches close to or surpasses expert-written drivers in many instances, demonstrating its versatility and effectiveness. Additionally, DAISY outperforms FuzzGen in 4 out of the 7 projects FuzzGen is compatible with.

Compared with IntelliGen, DAISY can achieve better block coverage. IntelliGen is a well-performing *library-based* driver synthesizer, however, its lack of object usage information results in synthesizing drivers that call interface functions without properly initialized arguments, hindering its effectiveness in covering more program logic code. Besides, IntelliGen fails to generate fuzz drivers for some projects in FuzzBench, because the main entry function of those projects requires at least one function pointer argument, whose value cannot be generated properly by IntelliGen. DAISY in contrast can

derive such information through the object usage information and perform parameter initialization, allowing fuzzers to pass through sanity checks and cover more program logic code.

DAISY performs even better than the expert-written drivers in some cases. Although DAISY mostly performs better than the comparison synthesizers, it still performs behind expert-written drivers on some projects. One possible reason is that DAISY calls the interface functions with randomly generated arguments, which may trigger bugs and make it difficult to attain a high coverage. For instance, in *freetype2*, DAISY detects a heap-buffer-overflow crash in *FT_New_Face* with abnormal arguments, as we will further demonstrate in Section V-C. Triggering this bug stops DAISY's driver's fuzzing process, thus hindering its coverage progress. The other reason is that the effectiveness of DAISY depends greatly on the quality of library consumers. However, some consumers in FuzzBench cannot provide informative object usage for DAISY. The main shortage of expert-written drivers is that its effectiveness depends on the prior knowledge and experience of its writer. Besides, the off-the-shelf initial seeds are custom for existing drivers. This makes drivers synthesized by DAISY hardly take advantage of these initial seeds. In summary, although DAISY cannot consistently outperform the expert-written drivers, it is still better than other state-of-the-art driver synthesis tools, no matter the number of compatible or the driver's basic block coverage.

C. Real-World Case Study

In this section, we introduce our practice of vulnerability discoveries in six extensively-fuzzed real-world libraries. After 24-hour testing, DAISY found 9 previously-unknown bugs. Three of them are assigned CVEs because of their severe security consequences. Fuzz drivers generated by FuzzGen and IntelliGen are both not able to find the CVEs in the latest version of the project. The complexity of *glibc* prevents FuzzGen from getting a complete dependence graph, thus FuzzGen fails to detect the bugs. IntelliGen generates a fuzz driver by calling a function and performs a "lazy-store" operation for all its arguments. However, these CVEs all require at most two function calls to trigger, hence IntelliGen is not able to detect them. Table III shows the details of the bugs. Notably, a developer commented "too bad that the (previous) fuzzer didn't catch this, so thanks a lot for your report" when fixing a bug reported by DAISY. Since these libraries have been heavily tested for several years [1], we believe that the result demonstrates DAISY is able to uncover bugs that expert-written drivers miss.

We use *freetype2* from Google's FuzzBench and *libaom* from the Android Open Source Project as examples to show how DAISY synthesizes fuzz driver programs and how the driver has the capability of discovering previously unknown bugs. Both of these libraries are widely used in industrial products, for instance, Android, Chrome, etc. After showing the vulnerability detection capability of DAISY, we also investigate the effectiveness of DAISY's *OUS* merging algorithm.

TABLE III: Previous-unknown Bugs detected by DAISY.

Project	Bug Type	Identifier
freetype2	Heap-Buffer-Overflow	CVE-2022-27404
freetype2	SIGSEGV	CVE-2022-27405
freetype2	SIGSEGV	CVE-2022-27406
freetype2	Out-of-Memory	Issue-1153
matio	Out-of-Memory	Issue-190
libaom	Memory-Leak	Issue-3334
libvpx	Out-of-Memory	Issue-1764
file	SigABRT	Issue-348
grok	Memory-Leak	Issue-314

1) Synthesizing a driver for freetype2: freetype2 is a software font engine that is designed to be small, efficient, highly customizable, and portable while capable of producing high-quality output (glyph images).

To generate a fuzz driver for *freetype2*, DAISY utilizes two consumers to synthesize a fuzz driver for *freetype2*. One is *test_afm*, which is included by *freetype2* itself. The other is an example program from *freetype2's* website [39], named *ftsample*. Listings 2 and 3 show the main logic of them.

```
1  int main(int argc, char **argv) {
2    FT_Library      library;
3    FT_StreamRec    stream;
4    FT_Error        error = FT_Err_Ok;
5    AFM_FontInfoRec fi;
6    FT_Init_FreeType(&lib);
7    ...
8
9    if (error = FT_Stream_Open(&stream, argv[1]))
10     goto Exit;
11
12   stream.memory = library->memory;
13   if (error = parse_afm(library, &stream, &fi))
14     goto Exit;
15   ...
16   FT_Stream_Close(&stream);
17
18 Exit:
19   FT_Done_FreeType(library);
20   return error;
21 }
```

Listing 2 The consumer *test_afm.c* for *freetype2*

```
1  int main (int argc, char** argv) {
2    FT_Library lib;
3    if (FT_Init_FreeType(&lib))
4      return 1;
5
6    FT_Face face;
7    if (FT_New_Face(lib, argv[1], 0, &face))
8      return 1;
9
10   FT_UInt idx = FT_Get_Char_Index(face, 0);
11   if (FT_Load_Glyph(face, idx, 0))
12     return 1;
13
14   if (FT_Render_Glyph(face->glyph, 0))
15     return 1;
16
17   FT_Done_FreeType(lib);
18 }
```

Listing 3 The consumer *ftsample.c* for *freetype2*

```
(Allocate) library, stream, error, fi, face, idx;
(Call) FT_Init_FreeType(&library);
[Branch 1 Start]
(Call) FT_Stream_Open(&stream, _0);
(Store) stream.memory = library->memory;
(Call) parse_afm(library, &stream, &fi);
(Call) FT_Stream_Close(&stream);
[Branch 2 Start]
(Call) FT_New_Face(library, _1, _2, &face);
(Call) idx = FT_Get_Char_Index(face, _3);
(Call) FT_Load_Glyph(face, idx, _4);
(Call) FT_Render_Glyph(face->glyph, 0);
[Branch 1 End]
[Branch 2 End]
(Call) FT_Done_FreeType(library);
```

Listing 4 The merged *OUS* for *freetype2*

```
#0 sfnt_init_face (stream=..., face=...,
     face_instance_index=-16711680)
#1 in tt_face_init (stream=..., ttface=...,
     face_index=-16711680, num_params=0, params=0x0)
#2 in open_face (driver=..., astream=...,
     external_stream=0 '\000', face_index
     =71776119044505600, num_params=0, params=0x0,
     aface=...)
#3 in ft_open_face_internal (library=..., args=...,
     face_index=71776119044505600, aface=...,
     test_mac_fonts=0 '\000')
#4 in FT_New_Face (library=..., pathname=...,
     face_index=71776119044505600, aface=...)
#5 in LLVMFuzzerTestOneInput (data=..., size=17)
```

Listing 5 The heap-buffer-overflow in *freetype2* detected by DAISY

Both consumers call the function *FT_Init_FreeType()* to initialize an *FT_Library* type context variable. The main difference is how they use the variable: *test_afm* uses it to read and parse an *afm* file, while *ftsample* uses it to create a font face variable and render its glyph. At last, both of the two consumers call *FT_Done_FreeType()* to free the variable. After executing the two consumers, DAISY merges the extracted *OUS*, which is shown in Listing 4. When merging the two *OUS*es, DAISY finds that they both call *FT_Init_FreeType()* initially and *FT_Done_FreeType()* finally, with the arguments passed to the two functions being identical. Therefore, DAISY links the records which call *FT_Init_FreeType()* or *FT_Done_FreeType()* together. This merged *OUS* indicates that *FT_Init_FreeType()* must be called at the beginning of the driver, and *FT_Done_FreeType()* must be called at the end.

Based on the merged *OUS*, DAISY generates a fuzz driver for *freetype2*. After fuzzing for about ten minutes, LibFuzzer reports a heap-buffer-overflow (CVE-2022-27404 was assigned to the bug) in function *FT_New_Face()*, as shown in Listing 5. Specifically, the fuzzer tries to call *FT_New_Face()* with a too-large value for the *face_index* argument, and it is truncated to a signed 32-bit integer in its low-level implementation, thus leading to the heap-buffer-overflow bug.

2) Synthesizing a driver for libaom: libaom is a video codec library from Google and the Alliance for Open Media (AOMedia). It serves as the reference software implementation for the AV1 video coding formats. The library provides several

```
int main (int argc, char** argv) {
  ...
  aom_codec_ctx_t codec;
  AvxVideoReader *reader =
    aom_video_reader_open(argv[1]);
  const AvxVideoInfo *info =
    aom_video_reader_get_info(reader);
  aom_codec_iface_t *decoder =
    get_aom_decoder_by_fourcc(info->codec_fourcc);
  aom_codec_dec_init(&codec, decoder, NULL, 0);

  while (aom_video_reader_read_frame(reader)) {
    aom_codec_iter_t iter = NULL;
    size_t frame_size = 0;
    const unsigned char *frame =
      aom_video_reader_get_frame(
        reader, &frame_size);
    aom_codec_decode(&codec,
      frame, frame_size, NULL);
    while (aom_codec_get_frame(&codec, &iter)) {...}
    ...
  }
  ...

  aom_codec_destroy(&codec);
  aom_video_reader_close(reader);
}
```

Listing 6 The library consumer *simple_decoder.c* for *libaom*

```
(Allocate) codec, iter, frame_size;
(Call) reader=aom_video_reader_open(_0);
(Call) info=aom_video_reader_get_info(reader);
(Call) decoder=get_aom_decoder_by_fourcc(_1);
(Call) _2=decoder->codec_interface();
(Call) aom_codec_dec_init(&codec, _2, NULL, 0);
(Call) aom_video_reader_read_frame(reader);
(Call) frame=aom_video_reader_get_frame(
         reader, &frame_size);
(Call) aom_codec_decode(&codec,
         frame, frame_size, NULL);
(Call) aom_codec_get_frame(&codec, &iter);
...
(Call) aom_codec_destroy(&decoder);
(Call) aom_video_reader_close(reader);
```

Listing 7 The *OUS* extracted from *libaom* library consumer

consumer programs, where we utilize DAISY to synthesize a fuzz driver based on the simplest consumer *simple_decoder.c*.

The main function of *simple_decoder.c* is shown in Listing 6. The consumer calls a series of API functions in a specific order. By executing the consumer, DAISY constructs an Object Usage Sequence as shown in Listing 7. *codec*, *iter*, and *frame_size* are three explicit objects; *_0*, *_1*, and *_2* in Listing 7 are three implicit objects. From the two listings, we can see that all object usage patterns are abstracted from the original consumer. Take the *codec* object as an example, it is allocated at Line 1 and used in Lines 6, 10, and 12, which is in the same usage order as the library consumer. DAISY then synthesizes a fuzz driver using the procedure described in Section IV-B.

After fuzzing for about one minute, LibFuzzer reports a memory-leak bug in *aom_video_reader_open*, as shown in Listing 8. If the checking function *file_is_obu* fails, *libaom* will return a null pointer from *aom_video_reader_open*. However, *file_is_obu* does not free all chunks allocated by itself, hence leading to the memory-leak bug.

```
#0 in malloc (size=512000)
#1 in file_is_obu (obu_ctx=...)
#2 in aom_video_reader_open (filename=...)
#3 in LLVMFuzzerTestOneInput (data=..., size=3)
```

Listing 8 The memory-leak detected in *libvpx* by DAISY

3) **The effectiveness of DAISY's *OUS* merging algorithm.** For the two projects, we take three other consumers, use DAISY to extract *OUS* from each consumer and synthesize a fuzz driver based on each single *OUS*. We run each fuzz driver 24 hours 5 times and take the highest code coverage as the indicator to judge the quality of the fuzz driver. Afterward, we merge the three *OUS*es into one and generate another driver based on the merged *OUS*, also run it 24 hours 5 times and present the highest code coverage.

TABLE IV: Highest basic block coverage over 5 runs in 24 hours using fuzz drivers generated based on different *OUS*es.

	freetype2	libaom
OUS1	1216	1141
OUS2	14017	15148
OUS3	16837	1310
Union	18858	15790
DAISY	20072	20933
Improvement	6.4%	31.1%

Tabel IV shows the final result. The first three rows show the three fuzz drivers' coverage. The fourth row shows the union coverage of the three fuzz drivers. Since the code coverage of the three fuzz drivers based on the single *OUS* overlaps with each other, the union code coverage is smaller than the simple sum of the three fuzz drivers' coverage. At the same time, DAISY also merges the three *OUS* into one, and synthesizes a fuzz driver based on the merged *OUS*, whose coverage is shown in the fifth row. We can see that DAISY covers 6.4% and 31.1% more codes than the union of three fuzz drivers based on the unmerged *OUS* on *freetype2* and *libaom*, respectively. This shows that DAISY's *OUS* merging algorithm can improve the code coverage of the generated fuzz driver.

In summary, DAISY extracts object usage information from real-world library consumers and generates driver programs, which allows the fuzz engine to detect previously-unknown bugs, even in those extensively-fuzzed real-world benchmarks.

D. Discussion

In this section, we discuss the limitations of DAISY. The first limitation is the dependency on consumers. DAISY generates fuzz drivers by extracting and merging *OUS*es from library consumers. Without consumers, DAISY cannot get the *OUS*, hence unable to generate a fuzz driver. For the same reason, DAISY cannot generate corner cases that are not included in the consumers. One promising solution is to automatically generate test cases for libraries as their consumers to assist fuzz driver synthesis. We leave this to future work. The second limitation is the insufficient exploration of the initial seeds. Meanwhile, as discussed in Section V-B, DAISY can hardly take advantage of the off-the-shelf initial seeds provided by the library, since they are provided for the existing fuzz drivers. Besides, 2 fuzz drivers generated by DAISY may introduce new bugs because of the merging *OUS*es. Currently, we merge all *OUS*es one by one, once the merged *OUS* leads to a false positive, we then discard the last *OUS* to avoid the bug.

VI. LESSONS LEARNED

The usability of drivers in practical settings will impact the efficiency of fuzzing. Generally speaking, driver synthesizers need to be relatively effective and versatile, i.e. allow the fuzzer to test a significant proportion of the code and can be used across various libraries. While previous attempts at automated driver synthesis have achieved varying degrees of improvement over their predecessors, they are not well-suited for practical settings. DAISY improves significantly in its effectiveness and versatility by utilizing dynamic object usage information and can be utilized to generate drivers for a wide range of libraries.

More readable fuzz drivers are able to assist testers in better testing the entire project. In some cases, testers need to make small modifications to the automatically generated fuzz drivers to introduce new features. This requires the automatically generated fuzz drivers to be well-readable. DAISY generates fuzz drivers in C++ programming language. Though the readability of the fuzz drivers synthesized by DAISY could still be improved, testers may spend some time reading and understanding the automatically generated fuzz drivers, and modifying them appropriately to customize the fuzzing process.

A more accurate *OUS* merging algorithm needs more analysis of program semantics. Currently, DAISY merges two different *OUS*es based on a greedy algorithm, which only merges the function call nodes that may match. The algorithm may not take into account the rich program semantics embedded in the *OUS*. In the future, we may split the *OUS* into small pieces, analyze the program semantics of the target library from these fragment *OUS* pieces, then use the program semantics as a basis to combine the resulting *OUS* pieces into a fuzz driver. Such an algorithm may take more time to run than the current algorithm, but it can introduce richer program semantic information. By introducing richer program semantics, the readability of the automatically generated fuzz driver can be improved, and it may be able to cover more code and detect more hidden bugs in the target library.

VII. CONCLUSION

Current automated fuzz driver synthesis approaches experience difficulties in producing drivers that contain valid object usage and interface function invocation sequences, reducing the efficiency of library fuzzing and hindering the use of fuzzing in industrial settings. In this paper, we propose DAISY, a driver synthesis framework that extracts actual object usage sequences dynamically from library consumers and uses this information to construct driver programs with valid object

usage and interface invocation sequences. We implemented DAISY using the LLVM compiler framework and evaluated it on the Android Open Source Program (AOSP) and FuzzBench libraries against the synthesizers FuzzGen and IntelliGen. DAISY covered much more basic blocks than FuzzGen and IntelliGen, respectively, and detected 9 previously-unknown bugs with 3 CVEs assigned. Furthermore, DAISY is better suited for practical settings and we will open-source it for public usage, as it can synthesize drivers for more libraries than FuzzGen and IntelliGen.

REFERENCES

[1] G. Research, "Oss-fuzz: Continuous fuzzing for open source software," 2016, https://github.com/google/oss-fuzz.

[2] ——, "Cluster fuzz document," 2017, https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md.

[3] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[4] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[5] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz'em all: Generic language processor testing with semantic validation," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, 2021.

[6] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[7] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.

[8] 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)*. IEEE, 2021, pp. 318–327.

[9] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[10] "Daisy artifacts," 2022, https://github.com/frokaikan/daisy_artifacts/.

[11] lcamtuf, "American fuzzy lop," 2013, https://lcamtuf.coredump.cx/afl/.

[12] G. Research, "Google honggfuzz," 2010, https://github.com/google/honggfuzz.

[13] ——, "Libfuzzer," 2017, https://llvm.org/docs/LibFuzzer.html.

[14] 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*, 2018, pp. 475–485.

[15] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 499–513. [Online]. Available: https://doi.org/10.1145/3319535.3363225

[16] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1967–1983. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

[17] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: extend fuzzing optimizations of single mode to industrial parallel mode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 809–814.

[18] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K. R. Choo, "Deepfuzzer: Accelerated deep greybox fuzzing," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[19] M. Wang, J. Liang, C. Zhou, Y. Jiang, R. Wang, C. Sun, and J. Sun, "RIFF: Reduced instruction footprint for coverage-guided fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 147–159. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/wang-mingzhe

[20] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 858–870.

[21] Y. Shen, H. Sun, Y. Jiang, H. Shi, Y. Yang, and W. Chang, "Rtkaller: State-aware task generation for rtos fuzzing," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–22, 2021.

[22] M. Wang, Z. Wu, X. Xu, J. Liang, C. Zhou, H. Zhang, and Y. Jiang, "Industry practice of coverage-guided enterprise-level dbms fuzzing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 328–337.

[23] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in smt solvers using blackbox mutational fuzzing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 701–712.

[24] H. Shi, R. Wang, Y. Fu, M. Wang, X. Shi, X. Jiao, H. Song, Y. Jiang, and J. Sun, "Industry practice of coverage-guided enterprise linux kernel fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 986–995.

[25] 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 *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2811–2828. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-cen

[26] H. Green and T. Avgerinos, "Graphfuzz: Library API fuzzing with lifetime-aware dataflow graphs," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1070–1081. [Online]. Available: https://doi.org/10.1145/3510003.3510228

[27] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 75–84.

[28] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: An evolutionary test approach for java," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 185–194.

[29] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[30] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically generating test cases for specification mining," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 243–257, 2011.

[31] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1393–1403.

[32] the LLVM Compiler Infrastructure, "llvm-cov - emit coverage information," 2021, https://llvm.org/docs/CommandGuide/llvm-cov.html.

[33] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[34] 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)*, 2019, pp. 1949–1966.

[35] B. Mathis, R. Gopinath, and A. Zeller, "Learning input tokens for effective fuzzing," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 27–37.

[36] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ics protocol," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[37] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[38] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 999–1010.

[39] freetype2, "Website of freetype2," 2013, https://freetype.org/freetype2/docs/index.html.