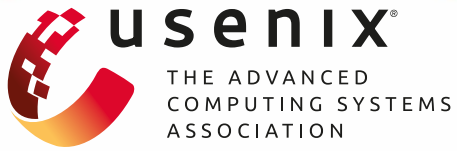

FuzzGen: Automatic Fuzzer Generation

Kyriakos Ispoglou, Daniel Austin, and Vishwath Mohan, *Google Inc.;*
Mathias Payer, *EPFL*

https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

FuzzGen: Automatic Fuzzer Generation

Kyriakos K. Ispoglou
Google Inc.

Daniel Austin
Google Inc.

Vishwath Mohan
Google Inc.

Mathias Payer
EPFL

Abstract

Fuzzing is a testing technique to discover unknown vulnerabilities in software. When applying fuzzing to libraries, the core idea of supplying random input remains unchanged, yet it is non-trivial to achieve good code coverage. Libraries cannot run as standalone programs, but instead are invoked through another application. Triggering code deep in a library remains challenging as specific sequences of API calls are required to build up the necessary state. Libraries are diverse and have unique interfaces that require unique fuzzers, so far written by a human analyst.

To address this issue, we present FuzzGen, a tool for automatically synthesizing fuzzers for complex libraries in a given environment. FuzzGen leverages a *whole system analysis* to infer the library's interface and synthesizes fuzzers specifically for that library. FuzzGen requires no human interaction and can be applied to a wide range of libraries. Furthermore, the generated fuzzers leverage LibFuzzer to achieve better code coverage and expose bugs that reside deep in the library.

FuzzGen was evaluated on Debian and the Android Open Source Project (AOSP) selecting 7 libraries to generate fuzzers. So far, we have found 17 previously unpatched vulnerabilities with 6 assigned CVEs. The generated fuzzers achieve an average of 54.94% code coverage; an improvement of 6.94% when compared to manually written fuzzers, demonstrating the effectiveness and generality of FuzzGen.

1 Introduction

Modern software distributions like Debian, Ubuntu, and the Android Open Source Project (AOSP) are large and complex ecosystems with many different software components. Debian consists of a base system with hundreds of libraries, system services and their configurations, and a customized Linux kernel. Similarly, AOSP consists of the ART virtual machine, Google's support libraries, and several hundred third party components including open source libraries and vendor specific code. While Google has been increasing efforts to fuzz test this code, e.g., OSS-Fuzz [35, 36], code in these repositories does not always go through a rigorous code review process. All these components in AOSP may contain vulnerabilities and could jeopardize the security of Android systems. Given the vast amount of code and its high complexity, fuzzing is a simple yet effective way of uncovering unknown vulnerabilities [20, 27]. Discovering and fixing new vulnerabilities is a crucial factor in improving the overall security and reliability of Android.

Automated generational grey-box fuzzing, e.g., based on AFL [44] or any of the more recent advances over AFL such as AFLfast [6], AFLGo [5], collAFL [19], Driller [37], VUzzer [31], T-Fuzz [28], QSYM [42], or Angora [8] are highly effective at finding bugs in programs by mutating inputs based on execution feedback and new code coverage [24]. Programs implicitly generate legal complex program state as fuzzed input covers different program paths. Illegal paths quickly result in an error state that is either gracefully handled by the program or results in a true crash. Code coverage is therefore an efficient indication of fuzzed program state.

While such greybox-fuzzing techniques achieve great results regarding code coverage and number of discovered crashes in *programs*, their effectiveness does not transfer to fuzzing *libraries*. Libraries expose an API without dependency information between individual functions. Functions must be called in the right sequence with the right arguments to build complex state that is shared between calls. These implicit dependencies between library calls are often mentioned in documentation but are generally not formally specified. Calling random exported functions with random arguments is unlikely to result in an efficient fuzzing campaign. For example, *libmpeg2* requires an allocated context that contains the current encoder/decoder configuration and buffer information. This context is passed to each subsequent library function. Random fuzzing input is unlikely to create this context and correctly pass it to later functions. Quite the contrary, it will generate a large number of false positive crashes when library dependencies are not enforced, e.g., the configuration function may set the length of the allocated decode buffer in the

internal state that is passed to the next function. A fuzzer that is unaware of this length field may supply a random length, resulting in a spurious buffer overflow. Alternatively, "invalid state checks" in library functions will likely detect dependency violations and terminate execution early, resulting in wasted fuzzing performance. To effectively fuzz libraries, a common approach is to manually write small programs which build up state and call API functions in a "valid" sequence. This allows the fuzzer to build up the necessary state to test functionality deep in the library.

libFuzzer [33] facilitates library fuzzing through the help of an analyst. The analyst writes a small "fuzzer stub", a function that (i) calls the required library functions to set up the necessary state and (ii) leverages random input to fuzz state and control-flow. The analyst must write such a stub for each tested component. Determining interesting API calls, API dependencies, and fuzzed arguments is at the sole discretion of the analyst. While this approach mitigates the challenge of exposing the API, it relies on *deep* human knowledge of the underlying API and its usage. Hence, this approach does not scale to many different libraries.

FuzzGen is based on the following intuition: *existing code* on the system *utilizes* the library in *diverse* aspects. Abstracting the graph of possible library dependencies allows us to infer the complex library API. Different aspects of the API are tested by *automatically generating custom fuzzer stubs* based on the inferred API. The automatically generated fuzzers will execute sequences of library calls that are similar to those present in real programs without the "bloat" of real programs, i.e., removing all computation that is not strictly necessary to build the state required for fuzzing. These fuzzers will achieve deep coverage, improving over fuzzers written by an analyst as they consider real deployments and API usage.

On one hand, many libraries contain unit tests that exercise simple aspects of the library. On the other hand, programs that utilize a library's API build up deep state for specific functions. Leveraging only individual test cases for fuzzing is often too simplistic and building on complex programs results in low coverage as all the program functionality is executed alongside the target library. Test cases are too simple and fail to expose deep bugs while full programs are too complex. A mechanism that automatically constructs arbitrarily complex fuzzer stubs with complex API interactions and library state allows sufficient testing of complex API functions. The set of all test cases and programs which use a library covers nearly all relevant API invocations and contains code to set up the necessary complex state to execute API calls. The vast amount of different library usages implicitly defines an *Abstract API Dependence Graph* (A^2DG). Based on this A^2DG it is possible to automatically create fuzzer stubs that test different aspects of a library (Figure 1).

To address the challenges of fuzzing complex libraries, we propose *FuzzGen*. FuzzGen consists of three parts: an API inference, an A^2DG construction mechanism, and a fuzzer

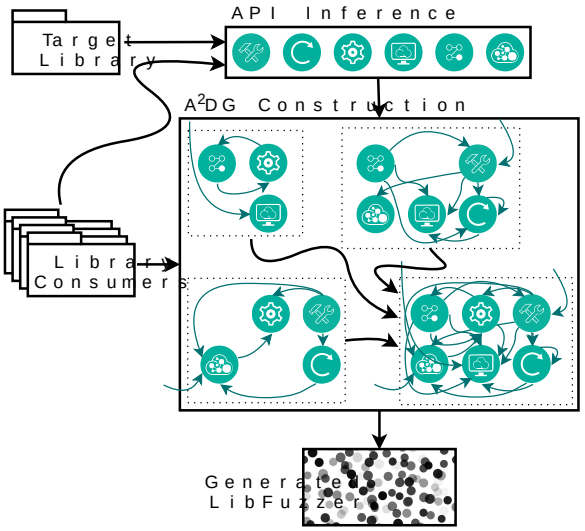

Figure 1: The main intuition behind FuzzGen. To synthesize a fuzzer, FuzzGen performs a whole system analysis to extract all valid API interactions.

generator that leverages the A^2DG to produce a custom libFuzzer "fuzzer stub". The API inference component builds an A^2DG based on all test cases and programs on a system that use a given library. The A^2DG is a graph that records all API interactions, including parameter value range and possible interactions. Our analysis infers library use and constructs a generic A^2DG based on this use. The fuzzer generator synthesizes fuzzers that build up complex state and leverage fuzz input to trigger faults deep in the library. FuzzGen automates the manual process of the analyst in creating custom-tailored fuzzers for libraries and specific library functions. The key contribution of FuzzGen is an *automatic* way to create new libFuzzer [33] stubs, enabling *broad and deep* library fuzzing.

FuzzGen performs a *whole system analysis*, iterating over all programs and libraries that use the target library to infer the A^2DG. It then *automatically* generates fuzzer stubs (ranging from $1,000$ to $10,000$ LoC) that encode the A^2DG and use libFuzzer to fuzz individual API components. FuzzGen was evaluated on Debian and Android [2].

Our evaluation of FuzzGen so far, resulted in 17 discovered vulnerabilities in the Android media framework, with 6 assigned CVEs: CVE-2019-2176 [16], CVE-2019-2108 [15], CVE-2019-2107 [14] and CVE-2019-2106 [13] (critical), CVE-2017-13187 [12] (high) and –duplicate– CVE-2017-0858 [11] (medium). (in Appendix C we provide more details on these vulnerabilities). Finding and eliminating vulnerabilities in these components is crucial to prevent potential attacks such as StageFright [17]. So far, FuzzGen has reported 17 new vulnerabilities in Android native libraries and Debian. The discovered bugs range from denial of service to stack buffer overflows, as shown in Section 6. Overall FuzzGen makes the following contributions:

- Design of a *whole system analysis* that infers valid API interactions for a given library based on existing programs and libraries that use the target library—abstracting the information into an Abstract API Dependence Graph (A^2DG);
- Based on the A^2DG, FuzzGen creates libFuzzer stubs that construct complex program state to expose vulnerabilities in deep library functions was developed—fuzzers are generated without human interaction;
- Evaluation of the prototype on AOSP and Debian demonstrates the effectiveness and the generality of the FuzzGen technique. Generating fuzzers for 7 libraries, FuzzGen discovered 17 bugs. The generated fuzzers achieve 54.94% code coverage on average, compared to 48.00% that fuzzer stubs—written manually by experts—achieve.

A note on disclosure: All bugs have been responsibly disclosed, and fixes have been pushed to the corresponding projects. The source code of our prototype is available at https://github.com/HexHive/FuzzGen, allowing other researchers to reproduce our results and to extend our automatic fuzzer generation technique.

2 The case for API-aware fuzzer construction

Writing an effective API-aware fuzzer requires an in-depth understanding of the target library and pinpointing the interesting components for fuzzing. Consider the libmpeg2 library, which provides encoding and decoding functions for MPEG2 video streams. The library contains several functions to build up a per-stream context that other functions take as a parameter. This approach of encapsulating state is common in libraries. Figure 2 shows a code snippet for properly initializing an MPEG2 decoding object. A fully initialized decoder object is required to decode a video frame. Without this decoder object, frames cannot be decoded.

While a target-agnostic fuzzer (invoking all functions with random arguments in a random order) may find simple issues, deep bugs will likely be missed due to their dependence on complex state. Naive fuzzers are also prone to false positives due to lack of API awareness. Consider a fuzzer that targets frame decoding. If the context does not contain a valid length with a pointer to an allocated decode buffer then the fuzzer will trigger a false positive crash when the decoded frame is written to unallocated memory. However, this is not a bug in the decode function. It is simply improper initialization. Orthogonally, by supplying random values to certain arguments, such as function pointers or sizes, a fuzzer may trigger memory errors. These crashes do not correspond to actual bugs or vulnerabilities as such an illegal context cannot be generated through any possible execution of a benign use of the library. Inferring API dependencies, such as generating a common

```
/* 1. Obtain available number of memory records */
iv_num_mem_rec_ip_t num_mr_ip = { ... };
iv_num_mem_rec_op_t num_mr_op = { ... };
impeg2d_api_function(NULL, &num_mr_ip, &num_mr_op);

/* 2. Allocate memory & fill memory records */
nmemrecs = num_mr_op.u4_num_mem_rec;
memrec   = malloc(nmemrecs * sizeof(iv_mem_rec_t));

for (i=0; i<nmemrecs; ++i)
    memrec[i].u4_size = sizeof(iv_mem_rec_t);

impeg2d_fill_mem_rec_ip_t fill_mr_ip = { ... };
impeg2d_fill_mem_rec_op_t fill_mr_op = { ... };
impeg2d_api_function(NULL, &fill_mr_ip, &fill_mr_op);

nmemrecs = fill_mr_op.s_ivd_fill_mem_rec_op_t
                     .u4_num_mem_rec_filled;

for (i=0; i<nmemrecs; ++i)
    memrec[i].pv_base = memalign(memrec[i].u4_mem_alignment,
  memrec[i].u4_mem_size);

/* 3. Initalize decoder object */
iv_obj_t *iv_obj = memrec[0].pv_base;
iv_obj->pv_fxns  = impeg2d_api_function;
iv_obj->u4_size  = sizeof(iv_obj_t);

impeg2d_init_ip_t init_ip = { ... };
impeg2d_init_op_t init_op = { ... };
impeg2d_api_function(iv_obj, &init_ip, &init_op);

/* 4. Decoder is ready to decode headers/frames */
```

Figure 2: Source code that initializes an MPEG2 decoder object. Low level details such as struct field initializations, variable declarations, or casts are omitted for brevity.

context, initializing the necessary buffers, and preparing it for usage, is challenging because dependencies are not encoded as part of the library specification.

However, by observing a module that utilizes libmpeg2 (i.e., a *library consumer*), we could observe the dependencies between the API calls and infer the correct order of context initialization calls. Such dependencies come in the form of (a) *control flow* dependencies and (b) *shared* arguments (variables that are passed as arguments in more than one API call). Furthermore, arguments that hold the state of the library (e.g., the context), should not be fuzzed, but instead they should be passed, without intermediate modification, from one call to the next. Note that this type of information is usually *not formally specified*. The libmpeg2 library exposes a single API call, `impeg2d_api_function`, that dispatches to a large set of internal API functions. Yet, this state machine of API dependencies is not made explicit in the code.

3 Background and Related Work

Early fuzzers focused on generating random parameters to test resilience of code against illegal inputs. Different forms of fuzzers exist depending on how they generate input, handle crashes, or process information. *Generational* fuzzers, e.g., PROTOS [32], SPIKE [1], or PEACH [18], generate inputs based on a format specification, while *mutational* fuzzers, e.g., AFL [44], honggfuzz [39], or zzuf [22], synthesize inputs through random mutations on existing inputs, according to some criterion (e.g., code coverage). Typically, increasing

code coverage and number of unique crashes is correlated with fuzzer effectiveness.

Mutational fuzzers have become the de-facto standard for fuzzing due to their efficiency and ability to adapt input. The research community developed additional metrics to classify fuzzers, based on their "knowledge" about the target program. *Blackbox* fuzzers, have no information about the program under test. That is, they treat all programs *equally*, which allows them to target arbitrary applications. *Whitebox* fuzzers are aware of the program that they test and are target-specific. They adjust inputs based on some information about the target program, targeting more "interesting" parts of the program. Although whitebox fuzzers are often more effective in finding bugs (as they focus on a small part of the program) and therefore have lower complexity, they require manual effort and analysis and allow only limited reuse across different programs (the whitebox fuzzer for program A cannot be used for program B). *Greybox* fuzzers attempt to find a balance between blackbox and whitebox fuzzing by inferring information about the program and feeding that information back to guide the fuzzing process. Evaluating fuzzers is challenging. We follow proposed guidelines [24] for a thorough evaluation.

Code coverage is often used in greybox fuzzers to determine if an input should be further evaluated. The intuition is that the more code a given input can reach the more likely is to expose bugs that reside deep in the code. Fuzzers are limited by the *coverage wall*. This occurs when the fuzzer stops making progress, and could be due to limitations of the model, input generation, or other constraints. Any newly generated input will only cover code that has already been tested. Several recent extensions of AFL have tried to address the coverage wall using symbolic or concolic execution techniques [23] and constraint solving. Driller [37] detects if the fuzzer no longer increases coverage and leverages program tracing to collect constraints along paths. Driller then uses a constraint solver to construct inputs that trigger new code paths. Driller works well on CGC binaries but the constraint solving cost can become high for larger programs. VUzzer [31] leverages static and dynamic analysis to infer control-flow of the application under test, allowing it to generate application-aware input. T-Fuzz [28] follows a similar idea but instead of adding constraint solving to the input generation loop, it rewrites the binary to bypass hard checks. If a crash is found in the rewritten binary, constraint solving is used to see if a crash along the same path can be triggered in the original binary. FairFuzz [26] increases code coverage by prioritizing inputs that reach "rare" (i.e., triggered by very few inputs) areas of the program, preventing mutations on checksums or strict header formats. FuzzGen addresses the coverage wall by generating multiple different fuzzers with different API interactions. The A^2DG allows FuzzGen to quickly generate alternate fuzz drivers that explore other parts of the library under test.

Although the aforementioned fuzzing approaches are effective in exposing unknown vulnerabilities, they assume that the target program has a well defined interface to supply random input and observe for crashes. These methods cannot be extended to deal with libraries. A major challenge is the *interface diversity* of the libraries, where each library provides a different interface through its own set of exported API calls. DIFUZE [10] was the first approach for interface-aware fuzzing of kernel drivers. Kernel drivers follow a well-defined interface (through `ioctl`) allowing DIFUZE to reuse common structure across drivers. FuzzGen infers how an API is used from existing use cases and generates fuzzing functions based on observed usage. SemFuzz [41], used natural-language processing to process the CVE descriptions and extract the location of the bug. Then it uses this information to synthesize inputs that target this specific part of the vulnerable code.

Developed concurrently and independently from FuzzGen, FUDGE [4] is the most recent effort on automated fuzz driver generation. FUDGE leverages a *single* library consumer to infer valid API usages of a library to synthesize fuzzers. However there are two major differences to our approach: *First*, FUDGE extracts sequences of API calls and their context (called "snippets") from a single library consumer and then uses these snippets to create fuzz drivers which are then tested using a dynamic analysis. Instead of extracting short snippets from consumers, FuzzGen *minimizes consumers* (iterating over the consumer's CFG) to only the library calls, their dependent checks, and dependent arguments/data flow. *Second,* FUDGE creates many small fuzz drivers from an extracted snippet. In comparison, FuzzGen merges *multiple* consumers to a graph where sequences of arbitrary length can be synthesized. Instead of the 1-N approach of FUDGE, FuzzGen uses an M-N approach to increase flexibility. Compared to FUDGE, FuzzGen fuzzers are larger, more generic, focusing on complex API interaction and not just short API sequences.

Beside fuzzing, there are several approaches to infer API usage and specification. One way to infer API specifications [29, 30] is through dynamic analysis. This approach collects runtime traces from an application, analyzes objects and API calls and produces Finite State Machines (FSMs) that describe valid sequences of API calls. This set of API specifications is solely based on dynamic analysis. Producing rich execution traces that utilize many different aspects of the library requires the ability to generate proper inputs to the program. Similarly, API Sanitizer [43] finds violation of API usages. APISan infers correct usages of an API from other uses of the API and ranks them probabilistically, without relying on whole-program analysis. APISan leverages symbolic execution to create a database of (symbolic) execution traces and statistically infers valid API usages. APISan suffers from limited scalability due to symbolic execution. As a static analysis tool, it may result in false positives. SSLint [21] targets SSL/TLS libraries and discovers API violations based on an analyst-encoded API graph. MOPS [7] is a static analyzer that uses a set of safe programming rules and searches for

violations of those rules. *Yamaguchi et. al* [40] present a technique that mines common vulnerabilities from source code, representing them as a code property graph. Based on this representation, they discover bugs in other programs.

4 Design

To synthesize customized fuzzer stubs for a library, FuzzGen requires both the library and code that exercises the library (referred to as library *consumer*). FuzzGen leverages a whole system analysis to infer the library API, scanning consumers for library calls. The analysis detects all valid library usage, e.g., valid sequences of API calls and possible argument ranges for each call. This information is essential to create reasonable fuzzer stubs and is not available in the library itself.

By leveraging actual uses of API sequences, FuzzGen synthesizes fuzzer code that follows valid API sequences, comparable to real programs. Our library usage analysis allows FuzzGen to generate fuzzer stubs that are similar to what a human analyst would generate after *learning the API* and *learning how it is used in practice*. FuzzGen improves over a human analyst in several ways: it leverages real-world usage and builds fuzzer stubs that are close to real API invocations; it is complete and leverages all uses of a library, which could be manually overlooked; and FuzzGen scales to full systems due to its automation without requiring human interaction.

At a high level, FuzzGen consists of three distinct phases, as shown in Figure 1. First, FuzzGen analyzes the target library and collects all code on the system that utilizes functions from this library to infer the basic API. Second, FuzzGen builds the *Abstract API Dependence Graph* (A^2DG), which captures all valid API interactions. Third, it synthesizes fuzzer stubs based on the A^2DG.

4.1 Inferring the library API

FuzzGen leverages the source files from the consumers to infer the library's exported API. First, the analysis enumerates all declared functions in the target library, $\mathcal{F}_{lib}$. Then, it identifies *all* functions that are declared in all included headers of all consumers, $\mathcal{F}_{incl}$. Then, the set of potential API functions, $\mathcal{F}_{API}$ is:

$$\mathcal{F}_{API} \leftarrow \mathcal{F}_{lib} \cap \mathcal{F}_{incl} \quad (1)$$

FuzzGen's analysis relies on the Clang framework to extract this information during the compilation of library and consumer. To address over-approximation of inferred library functions (e.g., identification of functions that belong to another library that is used by the target library), FuzzGen applies a *progressive library inference*. Each potential API function is checked by iteratively compiling a test program linked with the target library. If linking fails, the function is not part of the library. Under-approximations are generally not a problem as functions that are exported but never used in a consumer are not reachable through attacker-controlled code.

4.2 A^2DG construction

FuzzGen iterates over library consumers that invoke API calls from the target library and leverages them to infer valid API interactions. It builds an abstract *layout* of library consumers which is used to construct fuzzer stubs. Recall that FuzzGen fuzzer stubs try to follow an API flow similar to that observed in real programs to build up complex state. FuzzGen fuzzer stubs allow some flexibility as some API calls may execute in random order at runtime, depending on the fuzzer's random input. The A^2DG represents the complicated interactions and dependencies between API calls, allowing the fuzzer to satisfy these dependencies. It exposes which functions are invoked first (initialization), which are invoked last (tear down), and which are dependent on each other.

The A^2DG encapsulates two types of information: *control dependencies*, and *data dependences*. Control dependencies indicate how the various API calls should be invoked, while data dependencies describe the potential dependencies between arguments and return values in the API calls (e.g., if the return value of an API call is passed as an argument in a subsequent API call).

The A^2DG is a directed graph of API calls, similar to a coarse-grained Control-Flow Graph (CFG) that expresses sequences of valid API calls in the target library. Edges are also annotated with valid parameter ranges to further improve fuzzing effectiveness as discussed in the following sections. Each node in the A^2DG corresponds to a single call of an API function, and each edge represents *control flow* between two API calls. The A^2DG encodes the control flow across the various API calls and describes which API calls are reachable from a given API call. Figure 3 (a) shows an instance of the CFG from a libopus consumer. The corresponding A^2DG is shown in Figure 3 (b).

Building the A^2DG is two step process. First, a set of *basic* A^2DGs is constructed, one A^2DG for each root function in each consumer. Second, the A^2DGs of all consumers are coalesced into a single A^2DG.

Constructing a basic A^2DG. To build a basic A^2DG, FuzzGen starts with a consumer's CFG. If the consumer is a library, FuzzGen builds CFGs for each exported API function, otherwise it starts with the `main` function. To reconcile the collection of CFGs, FuzzGen leverages the *Call Graph* of the consumer. An individual analysis starts at the entry basic block of every *root* function in the call graph to explore the full consumer. This may lead to a large number of A^2DGs for a library consumer.

Starting from the entry basic block of a root function, FuzzGen iteratively removes *every* basic block that does *not* contain a `call` instruction to an API call. If a basic block contains multiple `call` instructions to API functions, the basic block is split into multiple A^2DG nodes with one API call each. If a basic block calls a non-API function, FuzzGen recursively calculates the A^2DG for the callee and results are integrated

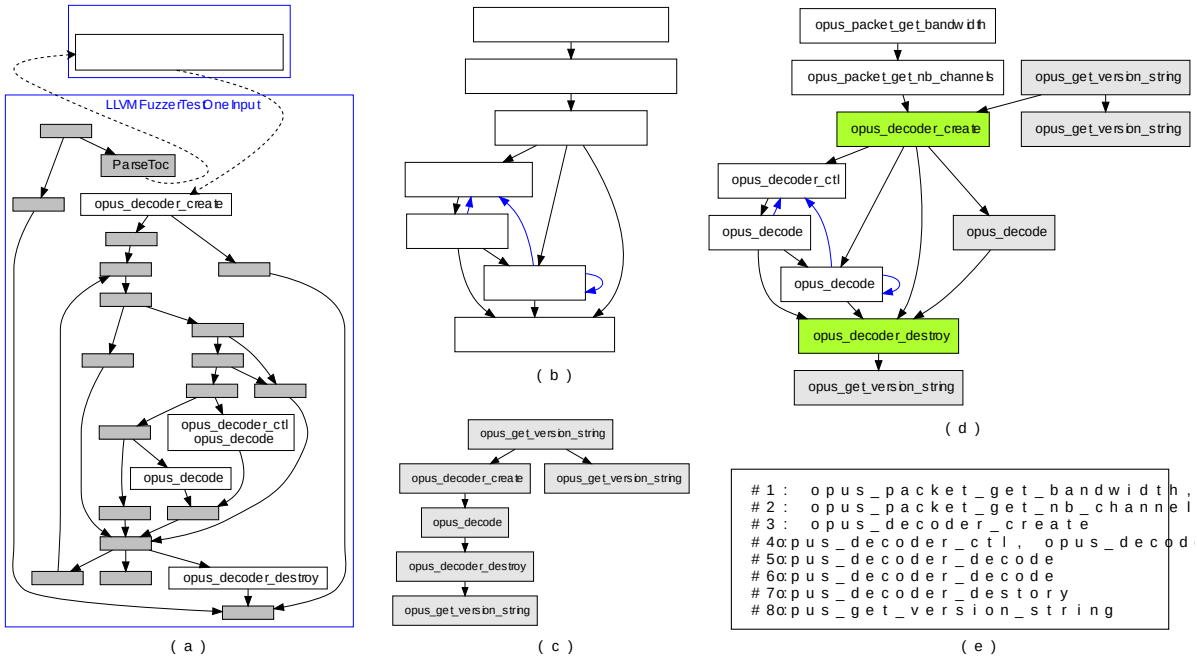

Figure 3: The FuzzGen workflow. FuzzGen starts with a CFG (a) and extracts the corresponding A^2DG (b) (see (c) for the graph of another module). The two A^2DG graphs are then merged (d). The merged A^2DG is then used to create fuzzers based on function orders (e). These graphs are autogenerated by FuzzGen.

into the caller's A^2DG. The pass integrates the calls into the root function. If the same non-API function is invoked multiple times, it is marked as a repeating function in the graph, avoiding an explosion of the graph's complexity. The algorithm to create the A^2DG is shown in Algorithm 1. A call stack (C_S) prevents unbounded loops when analyzing recursive functions. Two maps (M_{entry} and M_{exit}) link basic blocks to individual nodes in the A^2DG, allowing the algorithm to locate the A^2DG node a basic block corresponds to. Note that the only case where M_{entry} and M_{exit} are different is when a basic block contains more that one call to an API function.

After A^2DG construction, each node represents a single API call. The A^2DG allows FuzzGen to *isolate* the flows between API calls and expose their control dependencies. Basic A^2DG construction is a static analysis which results in some over-approximation during CFG construction due to indirect function calls. FuzzGen uses an LLVM *Link Time Optimization* (LTO) analysis pass to extract this information.

Coalescing A^2DG graphs. After generating A^2DGs for each consumer, FuzzGen merges them into a single A^2DG:

Select any two A^2DG graphs and try to coalesce them together. Repeat this process until there are no two A^2DG that can be coalesced together.

To coalesce two A^2DGs they **must** have at least one node in common. Two nodes are considered "common" if they invoke the same API call with the same arguments of the same type. FuzzGen starts from the root and selects the first common node. FuzzGen then removes the node from one graph and migrates all children, along with their sub trees, to the other A^2DG, continuously merging common nodes. A common node is a requirement, as placing the nodes from the second A^2DG at random positions will likely result in illegal target states. If there are no common nodes, FuzzGen keeps the A^2DGs separate, synthesizing two different fuzzers.

Figure 3 (d) shows an example of the A^2DG produced after coalescing the two A^2DGs in Figure 3 (b) and (c). The nodes with function `opus_decoder_destroy` are coalesced (as the argument is a handle, which has the same type), but other nodes like `opus_decoder_ctl` are not coalesced as the arguments are different. It is possible for the coalesced A^2DG to result in an *inconsistent state*, which results in an API *misuse*. That is, the coalesced A^2DG may contain a path (i.e., a subset of API calls) that violates API usage and therefore causes problems to execution state of the library. In Appendix A, we explain this problem in detail.

Our experiments showed that it may be feasible to coalesce two A^2DGs without common nodes by backward-slicing and locating function usages that invoke the API call. We leave this along with other heuristics to coalesce A^2DGs into a single one, for future work.

Precision of A^2DG construction. The current FuzzGen A^2DG construction has two sources of imprecision: static analysis and merging. First, the static analysis results in an over-approximation of paths. This may result in false posi-

Algorithm 1: A^2DG construction.

Input: Function F to start A^2DG construction
Output: The corresponding A^2DG

Function make_AADG(Function F)
  ▷ "$A \cup_= B$" is shorthand for "$A = A \cup B$"
  if $F \in C_S$ **then return** $(\emptyset, \emptyset)$ **else** $C_S \cup_= \{F\}$
  $G_{A^2DG} \leftarrow (V_{A^2DG}, E_{A^2DG})$
  foreach *basic block* $B \in CFG_F$ **do**
    ▷ An *empty* vertex is not associated with an API call
    Create *empty* vertex u, $V_{A^2DG} \cup_= \{u\}$,
     $M_{entry}[B] \leftarrow u$
  $Q \leftarrow \{entry\_block(F)\}$ ▷ single entry point
  while Q *is not empty* **do**
    remove basic block B from Q
    $v \leftarrow M_{entry}[B]$
    foreach `call` *instruction* $c_i \in B$ *in* ***reverse*** *order* **do**
      if $c_i.callee \in \mathcal{F}_{API}$ **then**
        if v *is empty* **then**
          $v \leftarrow c_i$, $M_{entry}[B] \leftarrow v$, $M_{exit}[B] \leftarrow v$
        else
          ▷ if already exists, split node
          $u \leftarrow c_i$
          $V_{A^2DG} \cup_= \{u\}$, $E_{A^2DG} \cup_= \{(u,v)\}$
          $v \leftarrow u$, $M_{entry}[B] \leftarrow u$
      else
        $AADG' \leftarrow$ make_AADG(c_i)
        Create *empty* vertex *sink*
        $V_{A^2DG} \cup_= V_{A^2DG'} \cup \{sink\}$
        $E_{A^2DG} \cup_= E_{A^2DG'}$
        foreach *leaf* $v_l \in AADG'$ **do**
          $E_{A^2DG} \cup_= \{(v_l, sink)\}$
        foreach *root* $v_r \in AADG'$ **do**
          $E_{A^2DG} \cup_= \{(v, v_r)\}$
    foreach *unvisited successor block* B_{adj} *of* B **do**
      add B_{adj} to Q
      $E_{A^2DG} \cup_= \{(M_{exit}[B], M_{entry}[B_{adj}])\}$
  ▷ Drop empty nodes from AADG
  foreach *empty node* $v \in AADG$ **do**
    foreach *predecessor* p *of* v **do**
      foreach *successor* s *of* v **do**
        $E_{A^2DG} \cup_= \{(p,s)\}$
    remove v and its edges from V_{A^2DG}
  $C_S \leftarrow C_S - \{F\}$
  return G_{A^2DG}

tives due to illegal API sequences that do not occur in real programs. Second, the merging process may over-eagerly merge two A^2DGs with different or slightly different parameters, resulting in illegal API sequences. We will discuss these sources of false positives in Section 7.

4.3 Argument flow analysis

To create effective fuzzers, the A^2DG requires both control and data dependencies. To construct the *data* dependencies between API calls FuzzGen leverages two analyses: *argument value-set inference* (what values are possible) and *argument dependence analysis* (how are individual variables reused).

Argument value-set inference. Argument value-set inference answers two questions: *which* arguments to fuzz and *how* to fuzz these arguments. Supplying arbitrary random values (i.e., "blind" fuzzing) to every argument imposes significant limitations both in the efficiency and the performance of fuzzing. Contexts, handles, and file/socket descriptors are examples that result in large numbers of false positives. Supplying random values for a descriptor in an API call results in *shallow coverage* as there are sanity checks at the beginning of the function call. Some arguments present diminishing returns when being fuzzed. Consider an argument that is used to hold output, or an argument that is part of a `switch` statement. In both cases, a fuzzer will waste cycles generating large inputs, where only a few values are meaningful. To better illustrate this, consider a fuzzer for `memcpy`:

```
void *memcpy(void *dest, const void *src, size_t n);
```

Supplying arbitrary values to `n` makes it inconsistent with the actual size of `src`, which results in a segmentation fault. However this crash does not correspond to a real bug. Also, the fuzzer may invest many cycles generating random values for the `dest` argument, which is never read by `memcpy()` (please ignore the corner case of overlapping source and destination arguments for the sake of the example).

Our analysis classifies arguments into two categories according to their type: *primitive arguments* (e.g., `char`, `int`, `float`, or `double`) and *composite arguments* (e.g., pointers, arrays, or structs). The transitive closure of composite arguments are a collection of primitive arguments—pointers may have multiple layers (e.g., double indirect pointers), structures may contain nested structures, or arrays—and therefore they *cannot* be fuzzed directly. That is, they cannot be assigned a random (i.e., fuzz) value, upon the invocation of the API call but require type-aware construction. Consider an API call with a pointer to an integer as the first argument. Clearly, fuzzing this argument results in segmentation faults when the function dereferences a likely invalid pointer. Instead, the pointer should point to an integer. The pointed-to address can be safely fuzzed. FuzzGen performs a data-flow analysis in the target library for every function for every argument, to infer the possible values that an argument could get.

Argument dependence analysis. Data-flow dependencies are as important as control-flow dependencies. A fuzzer must not only follow the intended sequence of API calls but must also provide matching data flow. For example, after creating a context, it must be passed to specific API calls for further processing. If this does not occur, it will likely result in a violation of a state check or a spurious memory corruption.

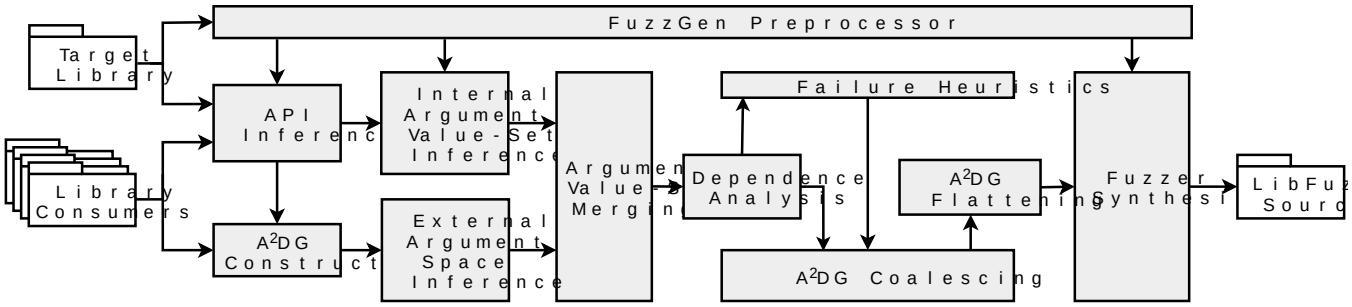

Figure 4: FuzzGen implementation overview.

Data-flow dependencies in an A^2DG can be intra-procedural and inter-procedural. First, FuzzGen identifies data dependencies through static per-function alias analysis of the code using libraries, tracking arguments and return values across API calls. Static alias analysis has the advantage of being complete, i.e., allowing any valid data-flow combinations but comes at the disadvantage of imprecision. For example, if two API calls both leverage a parameter of type `struct libcontext` then our static analysis may be unable to disambiguate if the parameters point to the same instance or to different instances. This over-approximation can result in spurious crashes. FuzzGen leverages backward and forward slicing on a per-method basis to reduce the imprecision due to basic alias analysis.

Second, FuzzGen identifies dependencies across functions: For each edge in the A^2DG, FuzzGen performs another data flow analysis for each pair of arguments and return values to infer whether whether they are dependent on each other.

Two alternative approaches could either (i) leverage concrete runtime executions of the example code which would result in an under-approximation with the challenge of generating concrete input for the runtime execution or (ii) leverage an inter-function alias analysis that would come at high analysis cost. Our approach works well in practice and we leave exploration of more precise approaches as future work.

The A^2DG (i.e., API layout) exposes the order and the dependencies between the previously discovered API calls. However, the arguments for the various API calls may expose further dependencies. The task of this part is twofold: *First*, it finds dependencies between arguments. For example, if an argument corresponds to a context that is passed to multiple consecutive API calls it should likely not be fuzzed between calls. *Second*, it performs backward slicing to analyze the data flow for each argument. This gives FuzzGen some indication on how to initialize arguments.

4.4 Fuzzer stub synthesis

Finally, FuzzGen creates fuzzer stubs for the different API calls and its arguments through the now complete A^2DG. An important challenge when synthesizing fuzzer stubs is to balance between depth and breadth of the A^2DG exploration. For example, due to loops, a fuzzer stub could continuously call the same API function without making any progress.

Instead of generating many fuzzer stubs for each A^2DG, FuzzGen creates a single stub that leverages the fuzzer's entropy to traverse the A^2DG. At a high level, a stub encodes all possible paths (to a certain depth) through the A^2DG. The first bits of the fuzzer input encode the path through the API calls of the A^2DG. Note that FuzzGen only encodes the sequence of API calls through the bits, not the complete control flow through the library functions themselves. The intuition is that an effective fuzzer will "learn" that if certain input encodes an interesting path, mutating later bits to explore different data-flow along that path. As soon as the path is well-explored, the fuzzer will flip bits to follow an alternate path.

5 Implementation

The FuzzGen prototype is written in about 19,000 lines of C++ code, consisting of LLVM/Clang [25] passes that implement the analyses and code to generate the fuzzers. FuzzGen generated fuzzers use libFuzzer [33] and are compiled with Address Sanitizer [34].

FuzzGen starts with a target library and performs a whole system analysis to discover all consumers of the library. The library and all consumers are then compiled to LLVM bitcode as our passes work on top of LLVM IR. Figure 4 shows a high level overview of the different FuzzGen phases.

The output of FuzzGen is a collection (one or more) of C++ source files. Each file is a fuzzer stub that utilizes libfuzzer [33] to fuzz the target library.

Target API inference. FuzzGen infers the library API by intersecting the functions implemented in the target library and those that are declared in the consumers' header files.

A^2DG construction. FuzzGen constructs a per-consumer A^2DG by filtering out all non-API calls from each consumer's CFG, starting from the root functions. For program consumers, the root function is `main`. To support libraries as consumers, root functions are functions with no incoming edges (using a backwards data-flow analysis to reduce the imprecision through indirect control-flow transfers).

Attribute	Description
dead	Argument is not used
invariant	Argument is not modified
predefined	Argument takes a constant value from a set
random	Argument takes any (random) value
array	Argument is an array (pointers only)
array size	Argument represents an array size
output	Argument holds output (destination buffer)
by value	Argument is passed by value
NULL	Argument is a NULL pointer
function pointer	Argument is a function pointer
dependent	Argument is dependent on another argument

Table 1: Inferable argument attributes from value-set analysis.

Internal Argument Value-Set inference. Possible values and their corresponding types for the function arguments are calculated through a per-function data flow analysis. FuzzGen assigns different attributes to each argument based on these observations. These attributes allow the fuzzer to better explore the data space of the library. Note that this process is imprecise due to aliasing. Table 1 shows the set of possible attributes. For example, if an argument is only used in a switch statement, it can be encoded as a set of *predefined* values. Similarly, if the first access to an argument is a write, the argument is used to *output* information. Arguments that are not modified (such as file descriptors or buffer lengths) receive the *invariant* attribute.

External Argument Value-Set inference. Complementing the internal argument value-set inference, FuzzGen performs a backward slice from each API call through all consumers, assigning the same attributes to the arguments.

Argument Value-Set Merging. Due to imprecision in the analysis or potential misuses of the library, the attributes of the arguments may differ. We need to carefully consolidate the different attributes for each argument when merging the attributes. Generally, FuzzGen's analysis is more accurate with external arguments. These arguments tend to provide real use-cases of the function. Any internal assignments that give concrete values, are used to complement the externally observed values. Value-set merging is based on heuristics and may be adjusted in future work.

Dependence analysis. Knowing the possible values for each argument is not enough, the fuzzer must additionally know when to reuse the same variable across multiple functions. The dependence analysis infers when to reuse variables and when to create new ones between function calls. FuzzGen performs a per-consumer data-flow analysis using precise intra-procedural and coarse-grained inter-procedural tracking to connect multiple API calls. While a coarse-grained inter-procedural analysis may result in imprecision, it remains tractable and scales to large consumers. The analysis records any data flow between two API functions in the A^2DG. Similarly to other steps, aliasing may lead to further imprecision.

Failure Heuristics. To handle some corner cases, FuzzGen uses a heuristic to discard error paths and dependencies. Many libraries contain ample error checking. Arguments are checked between API calls and, if an error is detected, the program signals an error. The argument analysis will detect theses checks as argument constraints. Instead of adding these checks to the A^2DG, we discard them. FuzzGen detects functions that terminate the program or pass on errors and starts the detection from there.

A^2DG Coalescing. After initial A^2DG construction, each consumer results in a set of at least one A^2DG. To create fuzzers that explore more state, FuzzGen tries to coalesce different A^2DG. Starting from an A^2DG node where an API call shares the exact same argument types and attributes, FuzzGen continuously merges the nodes or adds new nodes that are different. If the two graphs cannot be merged, i.e., there is a conflict for an API call then FuzzGen returns two A^2DGs. If desired, the analyst can override merging policies based on the returned A^2DGs. However, coalescing may combine an API call sequence that results in a *state inconsistency* (see Appendix A for an example). An analyst may optionally disable coalescing and produce a less generic fuzzer for each consumer. Although this approach cannot expose deeper dependencies, it increases parallelism, as different fuzzers can target different aspects of the library.

A^2DG Flattening. So far, the A^2DG may contain complex control flow and loops. To create simple fuzzers, we "flatten" the A^2DG before synthesizing a fuzzer. Our flattening heuristic is to traverse the A^2DG and to visit each API call at least once by removing backward edges (loops) and then applying a (relaxed) topological sort on the acyclic A^2DG to find a valid order for API calls. While a topological sort would provide a total order of functions (and therefore result in an overly rigid fuzzer), we relax the sorting. At each step our algorithm removes all API functions of the same order and places them in a group of functions that may be called in random order.

Fuzzer Synthesis. Based on a flattened A^2DG, FuzzGen translates nodes into API calls and lays out the variables according to the inferred data flow. The fuzzer leverages some fuzz input to decode a concrete sequence for each group of functions of the same order, resulting in a random sequence at runtime. Before compiling the fuzzer, FuzzGen must also include all the necessary header files. During the consumer analysis, FuzzGen records a dependence graph of all includes and, again, uses a topological sort to find the correct order for all the header files.

FuzzGen Preprocessor. The source code to LLVM IR translation is a lossy process. To include details such as header declarations, dependencies across header files, pointer arguments, array types, argument names, and `struct` names, FuzzGen leverages a preprocessor pass that records this information for later analysis.

	Library Information						Consumer Information					Final A^2DG			
	Name	**Type**	**Src Files**	**Total LoC**	**Funcs**	**API**	**Total**	**Used**	**Total LoC**	**Avg D_c**	**UAPI**	**Graphs**	**Coal.**	**Nodes**	**Edges**
Android	libhevc	video	303	113049	314	1	2	2	3880	0.002	1	10	5	29	58
	libavc	video	190	83942	581	1	2	2	4064	0.002	1	9	4	29	53
	libmpeg2	video	118	19828	179	1	2	2	4230	0.001	1	9	5	30	56
	libopus	audio	315	50983	276	65	23	4	1079	0.074	12	4	4	24	30
	libgsm	speech	41	6145	31	8	9	4	396	0.060	7	4	4	57	88
Deb	libvpx	video	1003	352691	1210	130	40	4	594	0.075	13	4	4	29	46
	libaom	video	955	399645	4232	86	39	4	491	0.106	17	4	4	40	51

Table 2: Codec libraries and consumers used in our evaluation. **Library Information**: **Src Files** = Number of source files, **Total LoC** = Total lines of code (without comments and blank lines), **Funcs** = Number of functions found in the library, **API** = Number of API functions. **Consumer Information**: **Total** = Total number of library consumers on the system, **Used** = Library consumers included in the evaluation, **Total LoC** = Total lines of code of all library consumers (without comments and blank lines), **Avg** D_c = Average consumer density, **UAPI** = Number of API functions used in the consumers. **Final** A^2DG: **Graphs** = Total number of A^2DGs, **Coalesced** = Number of nodes coalesced (same as the number of A^2DGs merges, since our algorithm uses a single node for merging), **Nodes**, **Edges** = Total number of nodes and edges (respectively) in the final A^2DG.

6 Evaluation

Evaluating fuzzing is challenging due to its inherent non-determinism. Even similar techniques may exhibit vastly different performance characteristics due to randomness of input generation. *Klees. et al* [24] set out guidelines and recommendations on how to properly compare different fuzzing techniques. Key to a valid comparison are (i) a sufficient number of test runs to assess the distribution using a statistical test, (ii) a sufficient length for each run, and (iii) standardized common seeds (i.e., a small set of valid corpus files in the right format).

Following these guidelines, we run our fuzzers five (5) times each (since results from a single run can be misleading), with twenty-four (24) hour timeouts. In the FuzzGen experiments, coverage tails off after a few hours with only small changes during the remainder of the test run (see Figure 5). Longer timeouts appear to have a negligible additional effect on our results.

The effectiveness of a fuzzer depends on the number of discovered bugs. However, code coverage is a complementing metric that reflects a fuzzer's effectiveness to generate inputs that cover large portions of the program. Performance is an orthogonal factor as more executed random tests broadly increase the chances of discovering a bug.

Due to the lack of extensive previous work on library fuzzing, we cannot compare FuzzGen to other automatic library fuzzers. As mentioned in Section 1, the primary method for library fuzzing is to (manually) write a fuzzer stub that leverages the libFuzzer [33] engine. We evaluate our FuzzGen prototype on AOSP and Debian. Evaluating and testing FuzzGen on two different systems demonstrates the ability to operate in different environments with different sets of library consumers. Additionally, we compare FuzzGen against libFuzzer stubs written by a human analyst. A second method is to find a library consumer (which is a standalone application) and use any of the established fuzzing techniques. We forfeit the second method as the selection of the standalone application will be arbitrary and highly influence the results. There is no good metric on how an analyst would select the "best" standalone application.

To compare FuzzGen, we select seven (7) widely deployed codec libraries to fuzz. There are two main reasons for selecting codec libraries. *First*, codec libraries present a broad attack surface especially for Android, as they can be remotely reached from multiple vectors as demonstrated in the StageFright [17] attacks. *Second*, codec libraries must support a wide variety of encoding formats. They consist of complex parsing code likely to contain more bugs and vulnerabilities.

We manually analyzed the API usage of each library and wrote manual fuzzer stubs for libhevc, libavc, libmpeg2, and libgsm. Luckily AOSP already provides manually written fuzzers libopus, libvpx, and libaom which we can readily use. Some libraries such as libmpeg2 have complicated interface (see Section 2) and it took several weeks to sufficiently understand all libraries and write the corresponding fuzzer stubs. In comparison, FuzzGen generates a fuzzer in a few minutes given the LLVM IR of the library and the consumers.

Table 2 shows all libraries that we used in the evaluation for AOSP and Debian. Note that the libhevc, libavc, and libmpeg2 libraries have a single API call (see Figure 2 for an example) that acts as a dispatcher to a large set of internal functions. To select the appropriate operation, the program initializes a `command` field of a special `struct` which is passed to the function. Such dispatcher functions are challenging for fuzzer synthesis and we chose these libraries to highlight the effectiveness of FuzzGen.

6.1 Consumer Ranking

When synthesizing fuzzers, methods for consumer selection are important. Fuzzers based on more consumers tend to include more functionality. This functionality, represented by new API calls and transitions between API functions, can

Library	Manual fuzzer information								FuzzGen fuzzer information								Difference		
	Total LoC	Edge Coverage (%)				Bugs Found		exec/ sec	Total LoC	Edge Coverage (%)				Bugs Found		exec/ sec	p	Cov	Bugs
		Max	Avg	Min	Std	T	U			Max	Avg	Min	Std	T	U				
libhevc	308	56.15	55.70	55.32	0.32	2493	23	83	1170	74.50	74.16	74.01	0.18	404	7	29	0.012	+18.46	-16
libavc	306	54.91	50.30	44.71	4.28	283	1	*8	1155	70.62	65.98	64.65	2.33	0	0	151	0.008	+15.68	-1
libmpeg2	457	51.39	49.59	45.42	2.14	1509	3	20	1204	56.95	56.60	56.26	0.26	6753	3	47	0.012	+7.01	0
libopus	125	15.85	15.71	15.16	0.27	0	0	174	624	39.99	35.22	32.63	3.08	110	3	218	0.012	+19.51	+3
libgsm	121	75.55	75.55	75.31	0.00	0	0	5966	490	69.40	68.20	67.40	0.77	229	1	4682	0.012	-7.35	+1
libvpx	122	54.79	54.13	53.61	0.49	0	0	63	481	52.17	50.99	48.05	1.52	464652	1	2060	0.012	-3.14	+1
libaom	69	44.54	35.03	30.40	5.12	57	2	111	1132	41.10	33.43	25.96	5.87	75	2	166	0.674	-1.60	0

Table 3: Results from fuzzer evaluation on codec libraries. We run each fuzzer 5 times. **Total LoC** = Total lines of fuzzer code, **Edge Coverage** % = edge coverage (**max**: maximum covarage from best run, **avg**: average coverage across all runs, **min**: maximum coverage from the worst run, **std**: standard deviation of the coverage), **Bugs found** = Number of total (**T**) and unique (**U**) bugs found, **exec/sec** = Average executions per second (from all runs), **Difference** = The difference between FuzzGen and manual fuzzers (p value from Mann-Whitney U test, unique bugs and maximum edge coverage). *The executions per second in this case are low because all 283 discovered bugs are timeouts.

increase the fuzzer's complexity. An efficient fuzzer must take both the amount of API calls and the underlying complexity into account. It is important to consider how much initialization state should be constructed before fuzz input is injected into the process. It is also important to determine how many API calls should be used in a single fuzzer to target a particular aspect of the library. During the evaluation, we observed that adding certain consumers increased A^2DG complexity without increasing the API diversity or covering new functionality. Merging too many consumers increases A^2DG complexity without resulting in more interesting paths. Adding other consumers lead to the *state inconsistency* problem. Restricting the analysis to few consumers resulted in a more representative A^2DG, but opens another question: which set of consumers provide a representative set of API calls?

FuzzGen ranks the "quality" of consumers from a fuzzing perspective and creates fuzzers from high quality consumers. The intuition is the number of API calls per lines of consumer code (i.e., the fraction of API calls in a consumer) correlates to a relatively high usage of the target API. That is, FuzzGen selects consumers that are "library oriented". We empirically found that using four consumers demonstrates all features of our prototype, such as A^2DG coalescing, and results in small fuzzers that are easy to verify. For the evaluation, the generated fuzzers are manually verified to not violate the implicit API dependencies or generate false positives.

However, in Appendix B we demonstrate how the number of consumers affects the set of API functions and how the generated A^2DGs participate in the fuzzer. The number of applied consumers *tail off* at a certain point. That is, additional consumers increase fuzzer complexity without adding new "interesting" coverage of the API. In future work we plan to explore other heuristics or even random selections of consumers to construct potentially more precise A^2DGs.

Formally, our heuristic for ranking consumers, is called *consumer density*, D_c, and defined as follows:

$$\mathcal{D}_c \leftarrow \frac{\#\ distinct\ API\ calls}{Total\ lines\ of\ real\ code} \tag{2}$$

6.2 Measuring code coverage

Code coverage is important both as feedback for the fuzzer during execution and to compare different fuzzers' ability to explore a given program. Code overage can be measured at different granularities: function, basic block, instruction, basic block edges, and even lines of source code. FuzzGen, like AFL and libFuzzer, uses basic block edge coverage.

For the evaluation, FuzzGen uses SanitizerCoverage [9], a feature that is available in Clang. During compilation, SanitizerCoverage adds instrumentation functions between CFG edges to trace program execution. To optimize performance, SanitizerCoverage does not add instrumentation functions on *every* edge as many edges are considered redundant. This means that the total number of edges that are available for instrumentation during fuzzing do not correspond to the total number of edges in the CFG.

Measuring code coverage for a single fuzzing run may be misleading [24]. To address this, statistical testing is conducted across the five runs to calculate the *average* coverage over time. Since new code paths are found at different times, we cannot simply calculate the average coverage for a given time. To overcome this problem we use *linear interpolation* to approximate the coverage for each fuzzing run at given time intervals. Then we calculate the average code coverage on each interval and visualize it in Figure 5 and Figure 6. Finally, we also report the results of the Mann-Whitney U test, comparing the maximum coverage across all runs for manual/generated fuzzers, according to [3]. The coverage of the FuzzGen generated fuzzers are better ($p < 0.05$, two-tailed) than the manually written fuzzers for all libraries except for libaom, where the result is not statistically significant.

6.3 Android evaluation

To evaluate FuzzGen on Android, we set up a small device cluster (details are shown in Appendix D) to fuzz the first five (5) libraries. Table 3 shows the results of our fuzzing execution. The first observation is that manual fuzzers are

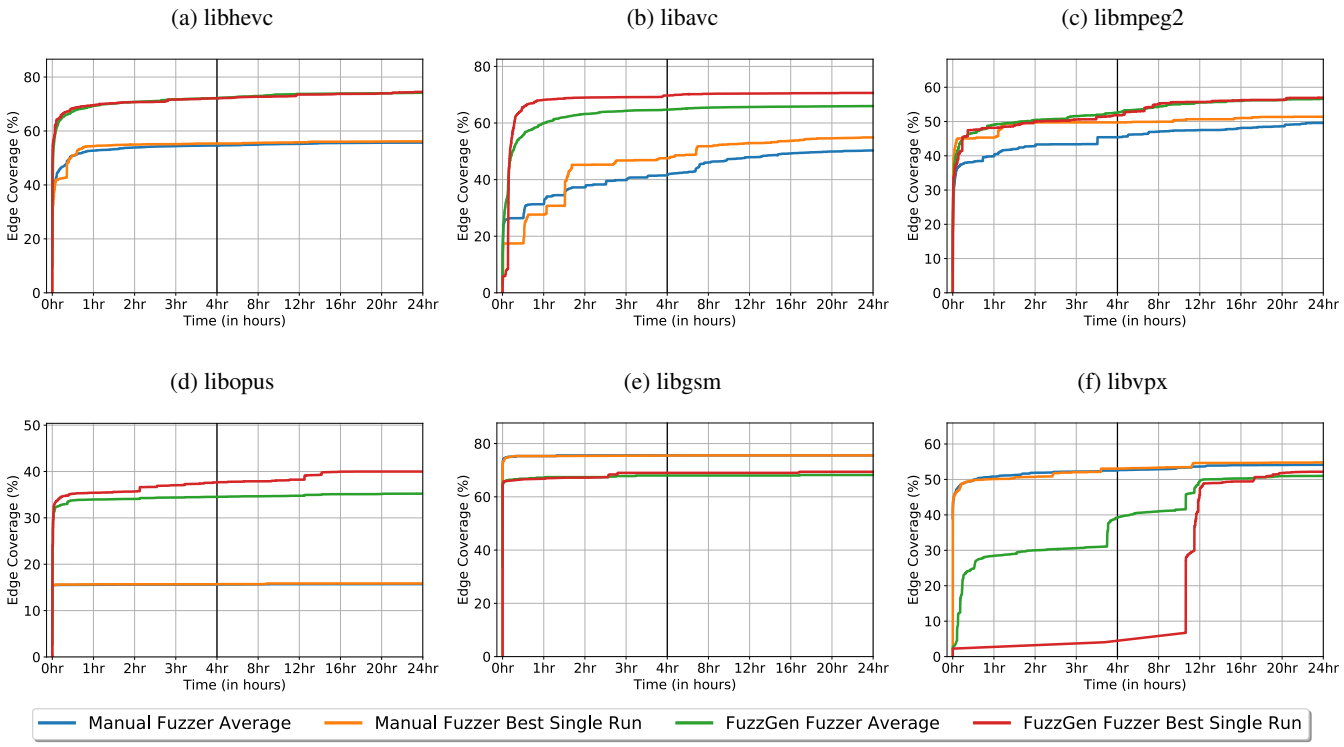

Figure 5: Code coverage (%) over time for each library. Blue line shows the average edge coverage over time for manual fuzzers and orange line shows the edge coverage for the best single run (among the 5) for manual fuzzers. Similarly, the green line shows the average edge coverage for FuzzGen fuzzers, and the red line the edge coverage from best single run for FuzzGen fuzzers.

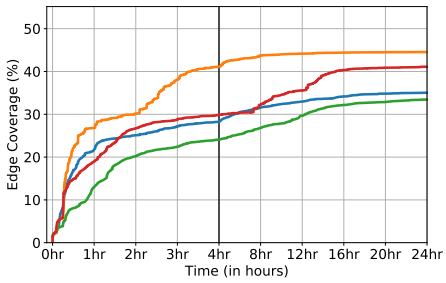

Figure 6: Code coverage over time for libaom.

smaller in size as they target only a specific part of the library (e.g., a decoding routine). Second, manual fuzzers are more targeted. Due to the focus on a single component, manual fuzzers can expose more bugs in that component compared to FuzzGen fuzzers. Meanwhile, FuzzGen fuzzers are broader and achieve a higher code coverage as they encode more diverse API interactions. This however imposes complexity which results in performance overhead, reducing the number of executions per second. Given more additional resources, FuzzGen fuzzers therefore allow the exploration of a broader program surface in the long run.

To get a better intuition on the evolution of the fuzzing process, we visualize the edge coverage over time as shown in Figure 5a through Figure 5e. The libopus library has lower total coverage (39.99%) than the other libraries tested. This is because all selected consumers focused on the decoding functionality. This aspect is highlighted in Table 2, where the fuzzer includes only 12 API calls while the API exposes 65 functions. A different selection of library consumers that utilize more aspects of the library (e.g., encoding functionality), would result in higher coverage, illustrating that selection of library consumers is crucial for FuzzGen fuzzing.

One of the key advantages of FuzzGen compared to manual fuzzer collection is the scalability and automation that FuzzGen provides. FuzzGen can leverage different sets of consumers to create many different fuzzers (e.g., a fuzzer stub focusing on encoding while another fuzzer stub focuses on decoding), allowing an analyst to explore different parts of the API in depth without having to manually create individual fuzzer stubs. These automatically created fuzzers can run at scale, simplifying the work of the analyst.

6.4 Debian evaluation

The Debian evaluation shows similar results to the Android evaluation. Two (2) additional codec libraries were selected for FuzzGen fuzzer generation. It is important to note the difference in library consumer selection. On Android, consumers are limited to those present in AOSP. On Debian, the

package manager is referenced to search for consumers that depend on the given library. In both cases we leverage a "finite" ecosystem where we can iterate through all available packages and select candidates to synthesize our fuzzers.

The last two columns of Table 3 show the results of running libvpx and libaom on Debian. The edge coverage over time is shown in Figure 5f and Figure 6 respectively. The first observation is that manual fuzzers have a lower rate of executions per second even though they are much smaller in size. This is because they contain *loops*. That is, they use a loop to continuously feed random input to the decoder. FuzzGen fuzzers are loop-free, which implies they spend less time on individual random inputs. For both libvpx and libaom, decoding multiple frames results in building a more complicated *decoding state*, which in turn triggers deeper code paths. It is better to have a fuzzer that contains loops for these cases, even though it achieves lower executions per second. For libopus, the decoding state is much simpler—since it is an audio library—so decoding multiple frames on each random input, affects performance, which results in a lower coverage.

7 Discussion and future work

Our prototype demonstrates the power of automatic fuzzer generation and API dependency analysis. As this is a first step towards automation, we would like to highlight several opportunities for improvement.

Maximum code coverage FuzzGen generated fuzzers achieve 54.94% coverage on average compared to manually written fuzzers that achieve only 48.00%. While FuzzGen vastly simplifies the generation of fuzzers, it remains an open question if the additional coverage substantially improves the fuzzing effectiveness in itself and if full cumulative coverage can be achieved by improving FuzzGen. The coverage we report is the cumulative coverage across all inputs in a single run. Given a fuzzer stub, only a certain amount of coverage can be achieved given through a combination of the used API functions and the arguments used for those functions. The problem of the maximum possible coverage that a fuzzer can achieve given a fuzzer stub is left for future work.

Single library focus. For now, FuzzGen focuses on a single target library and does not consider interactions between libraries. FuzzGen could be extended to support multiple libraries and interactions between libraries. This extension poses the interesting challenge of complex inter-dependencies but will allow the exploration of such inter-dependencies through an automated fuzzer. We leave this as future work.

Coalescing dependence graphs into a unifying A^2DG. When multiple library consumers are available, FuzzGen has to either *coalesce* all generated A^2DG into a single one or generate a separate fuzzer of each library consumer While the first approach can expose deeper dependencies and therefore achieve deeper coverage, it could potentially result in state inconsistencies. The latter approach increases parallelism, as different clusters can fuzz different aspects of the library.

False positives. Imprecision in the static analysis and the A^2DG coalescing may result in spurious paths that result in false positives. Fuzzing libraries is inherently challenging as the API dependencies are not known. The analysis could trace benign executions and extract benign API sequences to construct the A^2DG. This would result in an under-approximation of all valid API sequences. However, the static analysis combined with A^2DG coalescing results in an over-approximation. We argue that the over-approximation results in additional freedom for the fuzzer to generate more interesting path combinations, allowing FuzzGen to trigger deep bugs at the cost of a small false positive rate. In general, we propose to validate the API sequence during triaging. The analyst can trace the set of API calls and their parameters and manually check, for each reported crash, that the API sequence is valid. We empirically discovered that for some but few merged consumers, the likelihood of false positives is low. For our evaluation, we manually verified that the fuzzers cannot create false positives by double checking all API sequences.

8 Conclusion

Despite their effectiveness in vulnerability discovery, existing fuzzing techniques do not transfer well to libraries. Libraries cannot run as standalone applications and fuzzing them requires either a manually written libFuzzer stub that utilizes the library, or to fuzz the library through a library consumer. The wide diversity of the API and the interface of various libraries further complicates this task. To address this challenge, we presented FuzzGen, a framework that automatically infers API interactions from a library and synthesizes a target-specific fuzzer for it. FuzzGen leverages a whole system analysis to collect library consumers and builds an Abstract API Dependence Graph (A^2DG) for them.

We evaluated FuzzGen on 7 codec libraries —which are notorious for having a complicated interface— and in all cases, the generated fuzzers were able to discover 17 previously unknown vulnerabilities and received 6 CVEs.

The source code of our prototype is available online at https://github.com/HexHive/FuzzGen.

Acknowledgments

We thank our shepherd Tuba Yavuz and the anonymous reviewers for their insightful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 850868).

References

[1] Dave Aitel. An introduction to spike, the fuzzer creation kit. *presentation slides), Aug*, 2002.

[2] Open Handset Alliance. Android open source project, 2011.

[3] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[4] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei 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*, pages 975–985. ACM, 2019.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[7] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[8] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

[9] The clang development team: Sanitizer coverage. http://clang.llvm.org/docs/SanitizerCoverage.html, 2015.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[11] CVE-2017-0858: libavc: Another vulnerability in the android media framework. https://nvd.nist.gov/vuln/detail/CVE-2017-0858, 2017.

[12] CVE-2017-13187: libhevc: An information disclosure vulnerability in the android media framework. https://nvd.nist.gov/vuln/detail/CVE-2017-13187, 2017.

[13] CVE-2019-2106: libhevc: Stack-buffer-underflow in ihevc_sao_edge_offset_class2_chroma_ssse3. https://nvd.nist.gov/vuln/detail/CVE-2019-2106, 2019.

[14] CVE-2019-2107: libhevc: Multiple heap-buffer overflows in ihevcd_decode. https://nvd.nist.gov/vuln/detail/CVE-2019-2107, 2019.

[15] CVE-2019-2108: libhevc: Stack-buffer-overflow in ihevcd_ref_list, 2019.

[16] CVE-2019-2176: Heap buffer overflow in libhevcdec, 2019.

[17] Joshua Drake. Stagefright: Scary code in the heart of android. *BlackHat USA*, 2015.

[18] Michael Eddington. Peach fuzzing platform. *Peach Fuzzer*, 2011.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *CollAFL: Path Sensitive Fuzzing*, 2018.

[20] Patrice Godefroid. From blackbox fuzzing to whitebox fuzzing towards verification. In *Presentation at the 2010 International Symposium on Software Testing and Analysis*, 2010.

[21] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy (SP)*, 2015.

[22] S Hocevar. zzuf—multi-purpose fuzzer, 2011.

[23] James C King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[25] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[26] Caroline Lemieux and Koushik 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.

[27] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[28] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation, 2018.

[29] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *International Conference on Automated Software Engineering*, 2009.

[30] Michael Pradel and Thomas R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *International Conference on Software Engineering*, 2012.

[31] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[32] Juha Röning, M Lasko, Ari Takanen, and R Kaksonen. Protos-systematic approach to eliminate software vulnerabilities. *Invited presentation at Microsoft Research*, 2002.

[33] K Serebryany. libfuzzer: A library for coverage-guided fuzz testing (within llvm).

[34] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[35] Kostya Serebryany. Oss-fuzz. https://github.com/google/oss-fuzz.

[36] Kostya Serebryany. Oss-fuzz - google's continuous fuzzing service for open source software. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany, 2017.

[37] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[38] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, Thomas Wiegand, et al. Overview of the high efficiency video coding(hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 2012.

[39] Robert Swiecki. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz*, 2016.

[40] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, 2014.

[41] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[42] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: a practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[43] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API Usages through Semantic Cross-checking. In *Usenix Security Symposium*, 2016.

[44] Michal Zalewski. American fuzzy lop, 2015.

A State Inconsistency

Although coalescing increases the generality of the fuzzers, it suffers from the *state inconsistency* problem. Consider for instance a fuzzer of a socket library and two library consumers (a) and (b) as shown below:

```
/* consumer #1 */
sd = socket(...);
connect(...);

// send only sock
shutdown(sd,
         SHUT_RD);
write(sd, ...);

close(sd);
```

(a)

```
/* consumer #2 */
sd = socket(...);
connect(...);

// send & recv
write(sd, ...);

read(sd, ...);

close(sd);
```

(b)

```
/* coalesced */
sd = socket(...);
connect(...);

shutdown(sd,
         SHUT_RD);
write(sd, ...);
read(sd, ...);

close(sd);
```

(c)

The first module connects to a server and terminates the read side of the socket (as it only sends data). The second module both sends and receives data. If we ignore the arguments for now, the functions `socket`, `connect` and `write` are shared between the two consumers and they are therefore coalesced. The result is the coalesced fuzzer shown in (c). However this results in an inconsistency where the fuzzer closes the read side of the socket and later tries to read from it. Although the fuzzer does not crash, the coalesced module violates the state and is therefore not a useful fuzzer.

Consumers	API		A^2DG		
	Used	Found	Total	Nodes	Edges
0	0	0	1	0	0
1	6	34	1	7	12
2	6	34	1	9	14
3	10	34	1	16	22
4	12	34	1	24	30
5	25	51	1	142	289
6	31	51	2	148	303
7	33	65	2	181	438
8	44	65	1	540	1377
9	47	65	2	551	1393
10	50	65	2	611	1473
11	51	65	2	613	1475
12	53	65	2	697	1587
13	56	65	2	883	1773
14	56	65	2	885	1778
15	56	65	2	885	1778

Table 4: Complexity increase for the `libopus` library. **Consumers**: Total number of consumers used. **API**: **Used**: Total number of distinct API calls used in the final fuzzer. **Found**: Total number of distict API calls identified in headers. A^2DG: **Total**: Total number of A^2DG graphs produced (if coalescing is not possible there are more than one graphs). **Nodes** & **edges**: The total number of nodes and edges across all A^2DGs.

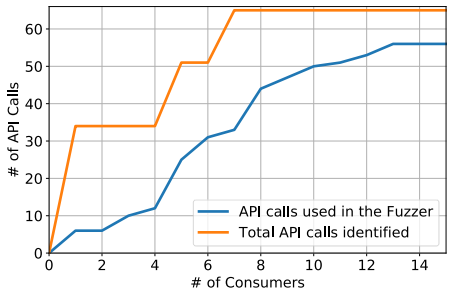

Figure 7: Consumer tail off for distinct API calls for `libopus` library.

A^2DG coalescing results in aggressive fuzzers that achieve deeper coverage and find more bugs. The downside is that coalescing may introduce false positives where the API is violated, resulting in false bugs. Without coalescing, the fuzzers are redundant and will not achieve coverage as deep as the coalesced fuzzers but will not introduce any false positives. In our empirical evaluation we discovered that the number of false positives is low and we therefore enable coalescing but leave it as a configurable option. In future work, we will look at how to tune coalescing aggressiveness, i.e., deciding how and when to coalesce based on a heuristic.

B Library Consumer Complexity

We empirically determined that a maximum of four consumers is a reasonable balance between complexity, breadth of explored API, and fuzzing effectiveness. To demonstrate the loss of effectiveness and the unnecessary increase in complexity when adding too may consumers, we present a case study on the `libopus` library where we continuously and iteratively merge consumers. We start with one consumer and progressively add more consumers (following our predetermined ranking). We measure the total number of API calls found in the consumer along with the size of the corresponding A^2DG. Table 4 shows how the number of consumers increases the size of the explored API. Only 7 consumers are enough to discover the complete API. However, the generated fuzzer only executes 33 different API calls. With increasing number of merged consumers, the fuzzer then executes more API calls until we reach a plateau at 13 merged consumers. Note that the fuzzer creates one path through the program that strings these API calls together. Libraries expose different functionality that are hard to streamline. This additional complexity slows down the fuzzer and prohibits it from discovering bugs quickly. Additionally, the generated fuzzers are harder for an analyst to analyze and, due to the repeated merging process, we increase the chances of false positives. Our observation is that it is better to create several smaller fuzzers than one complex fuzzer.

Figure 7 visualizes the discovery of API calls relative to the increasing set of merged library consumers. With 15 consumers FuzzGen generates a fuzzer stub with 8,375 lines of code. Despite this enormous size, it compiled and discovered a crash. However verifying whether this crash is a false positive or not, is extremely challenging due to the complexity of the API interactions in the fuzzer.

CVE number	Severity	Component	Description
CVE-2019-2176	Critical	libhevc	Heap Buffer Overflow in `ihevcd_parse_buffering_period_sei`
CVE-2019-2108	Critical	libhevc	Stack buffer overflow in `ihevcd_ref_list`
CVE-2019-2107	Critical	libhevc	Multiple heap buffer overflows in `ihevcd_decode`
CVE-2019-2106	Critical	libhevc	Stack buffer underflow in `ihevc_sao_edge_offset_class2_chroma_ssse3`
CVE-2017-13187	High	libhevc	Out of bounds read in `ihevcd_nal_unit`
CVE-2017-0858	Medium	libavc	NULL pointer dereference in `ih264d_parse_decode_slice`

Table 5: Assigned CVEs for the vulnerabilities found by FuzzGen.

C Overview of Disclosed Vulnerabilities

During our evaluation, the generated fuzzers discovered 17 vulnerabilities, 6 of which were assigned CVEs (Table 5). Following responsible disclosure, some vulnerabilities are still under embargo.

CVE-2019-2106 [13] is a *critical* vulnerability found in the High Efficiency Video Coding (HEVC) [38] library on Android. The vulnerability is an out of bounds write—which could lead to an arbitrary write—and resides inside `ihevc_sao_edge_offset_class2_chroma_ssse3`, as shown below:

```
void ihevc_sao_edge_offset_class2_chroma_ssse3(UWORD8 *pu1_src,
        WORD32 src_strd, UWORD8 *pu1_src_left, UWORD8 *pu1_src_top,
        UWORD8 *pu1_src_top_left, UWORD8 *pu1_src_top_right,
        UWORD8 *pu1_src_bot_left, UWORD8 *pu1_avail,
        WORD8 *pi1_sao_offset_u, WORD8 *pi1_sao_offset_v,
        WORD32 wd, WORD32 ht) {

    WORD32 row, col;
    UWORD8 *pu1_src_top_cpy, *pu1_src_left_cpy, *pu1_src_left_cpy2;

    /* ... */

    //availability mask creation
    u1_avail0 = pu1_avail[0];
    u1_avail1 = pu1_avail[1];
    au1_mask[0] = u1_avail0;
    au1_mask[1] = u1_avail0;
    au1_mask[wd - 1] = u1_avail1;
    au1_mask[wd - 2] = u1_avail1;  // Crash. OOB write --->
```

CVE-2017-13187 [12] is another *high severity* vulnerability found in the same component. This time, the vulnerability is an out of bounds read—which can cause remote denial of service—and resides inside `ihevcd_nal_unit` as shown below:

```
IHEVCD_ERROR_T ihevcd_nal_unit(codec_t *ps_codec)
{
    IHEVCD_ERROR_T ret = (IHEVCD_ERROR_T)IHEVCD_SUCCESS;

    /* NAL Header */
    nal_header_t s_nal;

    ret = ihevcd_nal_unit_header(&ps_codec->s_parse.s_bitstrm,
    &s_nal);

    RETURN_IF((ret != (IHEVCD_ERROR_T)IHEVCD_SUCCESS), ret);

    if(ps_codec->i4_slice_error)
        s_nal.i1_nal_unit_type =   // Crash. OOB read. --->
        ps_codec->s_parse.ps_slice_hdr->i1_nal_unit_type;
```

Supplying a frame with malformed slices to the decoder triggers both an out-of-bounds write (using the first vulnerability) and an out-of-bounds read (using the second vulnerability).

D Lab setup used for Android evaluation

To evaluate our fuzzers we utilized a set of twelve (12) Pixel-2 (walleye) devices. This setup allowed us to run repeated 24-hour fuzzing sessions. Figure 8 shows our device cluster.

Figure 8: Our device fuzzing cluster.