Invivo Fuzzing by Amplifying Actual Executions

Octavio Galland
MPI-SP, Germany

Marcel Böhme
MPI-SP, Germany

***Abstract*—A major bottleneck that remains when fuzzing software libraries is the need for *fuzz drivers*, i.e., the glue code between the fuzzer and the library. Despite years of fuzzing, critical security flaws are still found, e.g., by manual auditing, because the fuzz drivers do not cover the complex interactions between the library and the host programs using it.**

In this work we propose an alternative approach to library fuzzing, which leverages a valid execution context that is set up by a given program using the library (the *host*), and *amplify* its execution. More specifically, we execute the host until a designated function from a list of *target* functions has been reached, and then perform coverage-guided function-level fuzzing on it. Once the fuzzing quota is exhausted, we move on to fuzzing the next target from the list. In this way we not only reduce the amount of manual work needed by a developer to incorporate fuzzing into their workflow, but we also allow the fuzzer to explore parts of the library as they are used in real-world programs that may otherwise not have been tested due to the simplicity of most fuzz drivers.

I. Introduction

Today, fuzz drivers have to be developed to make a software library ammenable to fuzzing. *Fuzzing* is a popular automated testing technique which has proven to be very effective at bug finding, with tens of thousands of vulnerabilities discovered in commonly used software [1]. Since this technique involves executing the program, when applying it to code libraries it becomes necessary to implement a fuzz driver. A *fuzz driver* is a piece of code that acts as the entry point for execution during the fuzzing campaign. It sets up an artificial calling context and is responsible for accepting data from the fuzzer and feeding it into the library with the appropriate format.

Traditionally, fuzz drivers are implemented *manually* which constitutes a major hinderance to widespread adoption of fuzzing. For instance, Google incentivices the development of fuzz drivers for important open source projects by paying up to 30k USD of integration awards [2]. Google's project OSS-Fuzz [1] is primarily a community-maintained collection of fuzz drivers for open source projects.

Moreover, fuzz drivers do not capture the complex interaction that actual host programs have with the library. They often set up too simplistic a context and seperately target very specific parts of any given API. While this allows developers to maximize fuzzer throughput—for instance, LibFuzzer's documentation [3] advices developers to make fuzz drivers execute as fast as possible and leave any global state unmodified after execution has finished—it also reduces the search space for bugs that could be observed during normal execution by a host but not during executions generated via the fuzz drivers.

Amplifier Point	Amplifier Constraints
`ASN1_parse(` BIO* _, const char* *pp*, long *len*, int _)	sizeof(pp) = len $\wedge$ len $< C$
`OPENSSL_hexstr2buf(` const char* *str*, long* _)	sizeof(str) $< C$
`ossl_punycode_decode(` char* *pEnc*, size_t *encLen*, int* *pDec*, int* *pOutLen*)	sizeof(pEnc) = encLen $\wedge$ sizeof(pDec) = encLen $\wedge$ encLen $< C$ $\wedge$ sizeof(pOutLen) = 1 $\wedge$ *pOutLen = encLen
`cms_kek_cipher(` char** _, size_t* _, char* *in*, size_t *inlen*, CMS_KeyAgreeRecipientInfo* _, int _)	sizeof(in) = inlen $\wedge$ inlen $< C$

TABLE I: Four of 100+ amplifier points semi-automatically selected for OpenSSL. C is a constant chosen to prevent spurious out-of-memory errors. For brevity, we omit names for parameters which are not fuzzed (_).

In fact, despite an abundance of fuzz drivers and years of fuzzing, it is still possible to find vulnerabilities in software libraries by manual auditing that could have been found by fuzzing if only the "right" fuzz driver was written. For instance, recently a `high`-severity vulneratility has been found in the `OpenSSL` project which involved faulty memory management during parsing of `Punycode` and resulted in a stack-buffer overflow (CVE-2022-3602). In the aftermath of the discovery it was found that this vulnerability could be exposed in a matter of minutes through fuzzing, if only the relevant part of the code was being targeted.[1] Moreover, the relevant function was being executed by the test suite, which suggests that if the test suite had been "amplified" by fuzzing, the bug might have been spotted earlier.

In this paper, we propose *in-vivo fuzzing* to make all code subject to fuzzing by (i) identifying amplifier points, (ii) injecting function-level fuzzers at amplifier points, and (iii) amplifying actual executions from a host application that is using the target library. This approach side-steps any requirement for fuzz drivers and allows us to amplify executions generated in any way as they are entering an amplifier point.

Amplifier points. While it is possible to choose every function entry as amplifier point, to maximize utility we would like to focus on the most interesting functions. In order to identify such functions, the user can rely on their expert knowledge of the codebase, or on automated static analyses. For our implementation,[2] we choose functions that are associated with parsing a specific data chunk from the input sequence of bytes. Without loss of generality, this simplifies the data types used

[1] https://allsoftwaresucks.blogspot.com/2022/11/why-cve-2022-3602-was-not-detected-by.html

[2] https://anonymous.4open.science/r/afllive-598A/README.md#config-file-1

for amplification and minimizes the number of false positives. The criteria involves having at least one parameter whose type is a pointer to a byte-stream and has a parsing-related name (e.g., `parse`, `decode`). Table I-Col.1 shows four (of 100+) amplifier points automatically identified for OpenSSL.

Amplifier preconditions. In-vivo fuzzing mutates the input parameters of functions chosen as amplifier points. The hope is that the mutational approach corrupts the valid program state minimally to maintain the validity of the resulting program state and minimize the number of false positives. Nevertheless, there are certain *constraints* that need to be satisfied to maintain validity. These preconditions take the form of a conjunction of constraints that arguments need to satisfy. To minimize false positives, when specifying amplifier points, we allow such preconditions to be specified, as well. Table I-Col.2 shows the preconditions for the four amplifier points in OpenSSL.

Instrumentation and runtime. In order to fuzz a library with our approach it is first necessary to find a suitable host which uses it, and instrument both of them during compilation. This instrumentation enables us to intercept any call to an amplifier point during an execution of the host. Upon invocation of an amplifier point, we can proceed to *amplify* the execution by repeatedly forking into a *shadow execution*, replacing the parameters passed to the function with parameters provided by the fuzzer, and allowing this shadow execution to terminate while monitoring the shadow process for potential crashes.

In-vivo fuzzing (§III). To study the effectiveness of our in-vivo approach, we implemented our tool called AFLLIVE and measured the difference in code coverage achieved between the unamplified, original executions and the amplified, shadow executions. For each of the four target libraries, we chose a host program and input to generate unamplified executions. Indeed, we observe a substantial increase in code coverage without false positives, indicating that AFLLIVE effectively explores the "valid" neighborhood of the original execution.

Auto harnessing (§IV). An approach to enable fuzzing for a library without manual intervention is the automatic synthesis of fuzz drivers. To study the difference in effectiveness, we compare AFLLIVE against state-of-the-art fuzz driver generators. Specifically, we choose the fuzz drivers generated by *FuzzGen* [4] and *FUDGE* [5] and compare them with our approach on the FuzzGen and FUDGE benchmarks.[3] Ensuring the same initial conditions, we find that our prototype is able to identify 7 bugs in one of the subjects (5 memory corruption bugs and 2 assertion violations), while FuzzGen and FUDGE are unable to find any. Furthermore, our prototype consistently achieves greater code coverage on all of their benchmarks.

Test amplification (§V). We discuss the amplification of the executions generated by a test suite as a special use case of in-vivo fuzzing. For the OpenSSL example, amplifying the test suite did not only rediscover the `Punycode` vulnerability, but also found a previously unknown vulnerability that received

[3]We spent several months conducting experiments with the UTopia fuzz driver generator [6], but could only reproduce false positives on the original and most recent versions of the benchmark programs. Details in the Appendix.

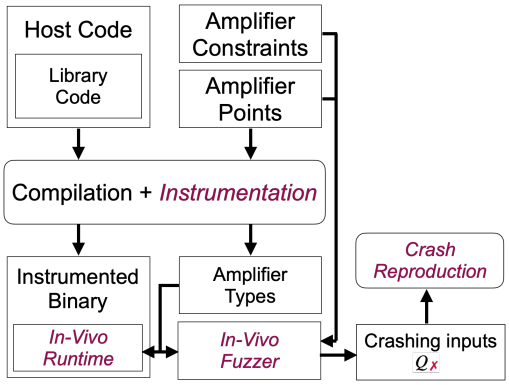


Fig. 1: Overall procedure.

2.4k USD in bug bounty. Test cases often cover certain edge cases or are designed to expose regressions similar to previously discovered bugs. Amplifying such test cases allows us to search their "neighborhood" to bring to light new bugs or existing bugs that have been incompletely fixed. In fact, over 40% of 0-days exploited in-the-wild are variants of previously discovered vulnerabilities.[4] Also, test suites often achieve high code coverage. Amplifying the test suite thus enables the fuzzer to reach deep into the code. During amplification, the test cases practically become fuzz drivers for in-vivo fuzzing.

In summary, the main contributions of this work are:

- An approach that auto-enables fuzzing for every compilable system or library subject to user-defined constraints that is executed, e.g., in production.
- A way to harness existing test suites by amplifying their coverage and exploring the neighborhoods states induced by existing regression tests.
- An open-source prototype implementation AFLLIVE and an extensive evaluation, available at: https://anonymous.4open.science/r/afllive-598A.

II. In-Vivo Fuzzing

Given a host process p and a set of interesting functions F, called amplifier points, *in-vivo fuzzing* piggybacks on the correct execution of the host process to generate a valid library state, calling context, and arguments for any function $f \in F$. Trying to minimize interference with the original process, an in-vivo fuzzer proceeds to repeatedly fork the execution and mutate the parameters of a targeted function call—within the constraints C specified by the user and in a coverage-guided manner—to generate a crashing function call.

In-vivo fuzzing is inspired by the mutational fuzzing approach for file-processing programs [7]–[11] or protocol implementations [12]–[15]. Given a valid seed input, such as a PDF file for a PDF reader, a *mutational fuzzer* slightly corrupts the valid file by applying various mutation operators to generate semi-valid files that can still reach deep into

[4]https://blog.google/threat-analysis-group/0-days-exploited-wild-2022/

the parsing process but suddenly induce crashes in the file-processing program. We might consider the generated inputs to be within the "neighborhood" of the valid seed file. Moreover, given a seed corpus with a high diversity, a mutational fuzzer can cover a large diversity of program behaviors in the file-processing program.

Algorithm 1 In-Vivo Fuzzing – Function `fuzz`

Input: Amplifier points F, Types T, Constraints C
Input: Instrumented process p, Time budget t_0 and t_1
1: Global corpus $Q = \emptyset$
2: Local corpus $Q_f = \emptyset$ for all $f \in F$
3: Crashes $Q_{\text{✗}} = \emptyset$
4: Shadow process $p' = \texttt{fork}(p)$
5: **for each** function $f \in F$ executed in p **do**
6: Objects *objs* = `collect_initial_args`(p', f)
7: Types $t \in T$ corresponding to f
8: Args *args* = `serialize`(*objs*, f, t)
9: Add *args* to local corpus Q_f
10: Add $\langle f, \textit{args} \rangle$ to global Q
11: **end for**
12:
13: **for each** function $f \in F$ executed in p **do**
14: **while** t_0 not expired **do**
15: Args q = `select`(Q_f)
16: For f, find types $t \in T$ and constraints $c \in C$
17: `fuzz_function_args`$(p', f, t, c, q, Q, Q_f, Q_{\text{✗}})$
18: **end while**
19: **end for**
20: **while** campaign not aborted and t_1 not expired **do**
21: Tuple $\langle f, q \rangle$ = `select`(Q)
22: For f, find types $t \in T$ and constraints $c \in C$
23: `fuzz_function_args`$(p', f, t, c, q, Q, Q_f, Q_{\text{✗}})$
24: **end while**
Output: Crashes $Q_{\text{✗}}$

Similarly, we propose to use as seed a *valid calling context* and *valid function arguments* generated by a host. This way an in-vivo fuzzer remains within the "neighborhood" of a valid program state when fuzzing a function. Assuming the host application generates several calls to the library at different amplifier points, our in-vivo fuzzer can reach deep into the program and cover and amplify a diverse set of program states.

A. Overall Procedure

Figure 1 sketches the overall procedure. Given the host code, including the library code, and the user-specified amplifier points and constraints, the first step is to compile and instrument the program. Our instrumentation pass introduces a function call into our in-vivo runtime within the preamble of every function identified as amplifier point. During fuzzing, this transfer of control allows the runtime to create a shadow process and independently fuzz the function arguments in that shadow process in collaboration with the in-vivo fuzzer. We implemented our prototype, AFLLIVE, on top of *AFL++ 4.02c*.

Instrumentation. Our instrumentation pass (LLVM 14 [16]) adds the runtime whose purpose is to mediate between the in-vivo fuzzer and running process of the instrumented binary. At the entry point of the main function, AFLLIVE inserts a call into our runtime to initialize any necessary state and communicate with the fuzzer. At the entry point of an amplifier point, the instrumentation pass inserts a call into our runtime containing the name of the amplifier point and the memory addresses of the arguments that will be fuzzed. Additionally, AFLLIVE hooks the exit point of every amplifier point to facilitate early termination of the host if configured in this way.

Amplifier points. To focus the in-vivo fuzzing on interesting library functions, we require the user to specify a set of functions called amplifier points. This selection need not be limited to library API functions only. It is possible to choose these amplifier points manually or using an auto-discovery process. Generally, we would be looking for functions whose signature or behavior suggests they may be attacker-controllable and contain vulnerabilities. For instance, our prototype uses CodeQL [17] to find parsing functions[5] (which are user-controlled by default and have the added benefit that constraint specification for them tends to be particularly simple, since they typically take a byte array and its length).

Amplifier constraints. To minimize the number of false positives, the in-vivo fuzzer also takes user-provided amplifier constraints that the generated function parameters need to satisfy before they are passed into the amplified function. These amplifier constraints carry the same role as the preconditions in property-based testing [18], [19]. They take the form of binary relationships between arguments and/or constants. For example, in Row 1 of Table I, a constraint can be seen which implies that the pointer `pp` must point to an array of length `len`, and that `len` should be less than a constant C.

Amplifier types. The instrumentation pass also records type information for the given amplifier points (in JSON format). During fuzzing, these amplifier types are used to serialize function parameter objects to a sequence of bytes for the in-vivo fuzzer and, vice versa, deserialize for the runtime, similar to the coverage-guided Java fuzzing approach proposed by Padhye, Lemieux, and Sen [19]. This type information is composed of the bitwidth for primitive types, fields' types and offsets for struct types, and the type of the pointee for pointers. Note that since the collected types are in LLVM Intermediate Representation (IR) types, these three cases cover every kind of variable types encountered for fuzzing most targets.

B. In-Vivo Fuzzing Algorithm

Algorithm 1 starts with the user-provided amplifier points and constraints, the auto-generated amplifier types, an instance of the instrumented host binary p, and two user-provided time budgets t_0 and t_1 (which determines the lengths of the screening phase and the main fuzzing loop, respectively).

Global and local corpora. In Line 1–3, all seed corpora are initially set to the empty set. The fuzzer maintains two global corpora Q and $Q_{\text{✗}}$ and one local corpus Q_f for every amplifier function $F \in F$. Throughout the campaign, the global corpus will contain *tuples* where the first element is

[5]This script checks each function's name against a predefined list of substrings and validates if at least one of its parameters is a pointer (or double pointer) of type `char*` or `uint8_t` and another one is an number.

an amplifier point and the second the serialized function arguments. A local corpus does not need the amplifier information and is hence a set of serialized function arguments. Since the fuzzer proceeds in a coverage-guided manner, the local and global corpora Q (and $Q_{\text{✗}}$) contain arguments that have been observed to be coverage-increasing (and crash-inducing, resp.).

Forking. In Line 4, the execution of process p is forked which allows the host to continue execution normally while we keep a handle to the shadow execution, which will be used for fuzzing. We assume that the forked process is isolated from the original execution and does not interfere with it. Conceptually, we assume the process has the ability to be rewound back to the invocation of any amplifier point, which is needed to alternate amplifier points during fuzzing.

Auto-collecting initial seeds. In Line 5–11, the fuzzer harvests the initial seeds from the original execution. These seeds are later used for coverage-guided, mutational fuzzing. For every amplifier point that is executed in the original, running process, our instrumentation pass made sure the call is routed through the in-vivo runtime which collects the function arguments as objects from the (forked) shadow process (Line 6). These objects corresponding to the function arguments are serialized and added to the global and local corpora.

Screening loop. In Line 12–18, the fuzzer fuzzes every executed amplifier point for a fixed amount of time in order to collect sufficient coverage information for the main fuzzing loop. The time budget t_0 for every amplifier point is fixed by the user. Without the screening loop, all amplifier points will be considered as equally good at generating coverage increasing inputs since the coverage collected during the initial, non-amplified execution will be exactly the same for all initial seeds. The screening loop over all amplifier functions forces the fuzzer to explore which regions of code each amplifier point is capable of covering. The function `select` (Line 14) selects the next best seed from the current local queue while the function `fuzz_function_args` (Line 16) fuzzes the selected seed. During fuzzing, all coverage-increasing inputs are added to the global and local queues (Q, Q_f) while all crashing inputs are added to the set of crashes ($Q_{\text{✗}}$).

Main fuzzing loop. In Line 19–23, the fuzzer fuzzes the seeds selected from the global queue until the campaign is aborted or the time budget t_1 is depleted. To select the next seed, we can now simply reuse the default heuristics of the underlying fuzzer (AFL++). Given a seed corpus of serialized function arguments, this is how we fuzz those arguments in a coverage-guided manner with a minimal false positive rate.

Early termination. AFLLIVE can be configured to terminate at the exit of or an arbitrary period of time after an amplifier function has returned, e.g., if the fuzzer throughput is too low. Our intuition is that crashes often arise shortly after the call to the amplified function. Early termination allows the user to strike a balance between performance and stability for a given campaign, at the risk of introducing false-negatives.

Algorithm 2 Function `fuzz_function_args`

Input: Process p', function f, types t, constraints c, args q
Input: Global corpus Q, Local corpus Q_f, Crashes $Q_{\text{✗}}$
1: Energy $e =$ `compute_energy`(f, q, Q)
2: **while** e not expired **do**
3: Mutated args $q' =$ `mutate`(q)
4: Mutated objs $o =$ `deserialize`(q', t, c)
5: Process $p'' =$ `fork_rewind_wait`(p', f)
6: Result $r =$ `substitute_continue`(p'', f, o)
7: **if** $r =$ new crash detected **then**
8: Add $\langle f, q' \rangle$ to $Q_{\text{✗}}$
9: **else if** $r =$ coverage increased **then**
10: Add $\langle f, q' \rangle$ to Q
11: Add q' to Q_f
12: **end if**
13: **end while**
Output: Global corpus Q, Local corpus Q_f, Crashes $Q_{\text{✗}}$

C. Mutational Fuzzing of Function Arguments

Algorithm 2 shows `fuzz_function_args` called in Line 16 and 22 of Algorithm 1. Given the shadow process, the amplifier point, types, and constraints, and the seed arguments, it mutates the seed to generate alternative function arguments. Those that increased code coverage are added to the local and global corpora while those that induced a unique crash are added to the set of crashes.

Mutation. In Line 1–3, an "optimal" number of mutations o' of the serialized function arguments q are created. What is considered as optimal is computed in the `compute_energy` function while the mutation operators applied to the arguments are implemented in the `mutate` function. Since the serialized arguments is just a sequence of bytes, we can reuse the implementations of both functions in a classic fuzzer [7], [20].

Deserialization. In Line 4, the in-vivo runtime receives and parses the mutated sequence of bytes into the actual function argument objects using the intrumenter-provided type information t. This process is deterministic: Deserializing the same byte sequence multiple times results in the same argument objects being generated, which ensures consistency and reproducibility. The deserialization procedure also enforces the user-provided constraints c. We discuss the procedure of `deserialize` in a separate section below.

Spawning shadow executions. In Line 5 and 6, the in-vivo runtime uses the shadow process p' to spawn another shadow execution which is rewound back to right before the selected function f is called, so as to continue executing with the mutated function parameters. Technically, we can implement the function `fork_rewind_wait` by considering p' as running in a virtual machine and using a snapshot-restore mechanism to restore a snapshot of p' right before f is called. This approach fully isolates the constructed process p'' from the shadow process p' (and the original process p), but it also introduces a performance and memory overhead for storing and loading the snapshots. In our prototype, we chose to fully reexecute the host application until f is reached (to conceptually rewind it) and start a fork server at f where the runtime interferes to provide the function call parameters o.

Algorithm 3 Function `deserialize`

Input: Function argument byte sequence q,
Input: Function argument types t
Input: Function argument constraints c
Output: Function argument objects o

```
 1: Objects o = ∅
 2: for type in t do
 3:     Object obj = deserialize_arg(q, type, c)
 4:     Add obj to o
 5: end for
 6: function deserialize_arg(q, type)
 7:     Object obj
 8:     if type is primitive then
 9:         obj = q.consume_bytes(type.bitWidth / 8)
10:     else if type = Struct then
11:         for field, fieldType in type.fields do
12:             obj.field = deserialize_arg(q, fieldType, c)
13:         end for
14:     else if type = Pointer then
15:         type' = type.pointeeType
16:         length = q.consume_bytes(4)
17:         for i ∈ {0, ..., length − 1} do
18:             obj[i] = deserialize_arg(q, type', c)
19:         end for
20:     end if
21:     Object obj = enforce_constraints(obj, c)
22:     return obj
23: end function
```

Coverage-guidance. In Line 7 to 12, the fuzzer adds function arguments to the corpora that are observed to be coverage-increasing. Function arguments that are observed to induce crashes are added to the corpus containing the crashing inputs.

D. Serialization and Deserialization

In order to reuse existing greybox fuzzers to implement seed selection (`select`), prioritization (`compute_energy`), and mutation (`mutate`), we need to translate function arguments into a sequence of bytes (which the fuzzer can handle) and back again. We accomodate the serialization (`serialize`) and deserialization (`deserialize`) procedures in the in-vivo runtime that is instrumented into the host binary (cf. Fig. 1). On a high-level, this process is similar to the appraoch proposed by Padhye et al. [19] which enables coverage-guided mutational fuzzing for an object-oriented language, like Java.

Algorithm 3 presents the procedure of the `deserialize` function. The procedure of `serialize` is analogous. Given a seed byte sequence, the argument types, and the argument constraints, the deserialization algorithm computes the function argument objects to be passed into the function call. In Line 1–5, the runtime generates one object for every function argument using its *type* information. Line 7–24 sketches the recursive procedure of the corresponding `deserialize_arg` function which also enforces the validity of the synthesized function argument objects.

The provided sequence of bytes q is "consumed", such that each byte is used at most once for the construction of the function argument objects. The function `consume_bytes` keeps an offset into the fuzzer-provided byte sequence, initially set to 0. When called, the function returns the desired amount of bytes available in the sequence, and advances the offset by that same amount of bytes. If the in-vivo runtime attempts to consume more bytes than available, the function returns all available bytes and the remaining bytes set to zero.

For *primitive types* (Line 9–10), the runtime reads as many bytes from the fuzzer-provided byte sequence q as needed in order to properly cast them into the appropriate type t.

For *structured types* (Line 11–14), an empty instance of the structure is allocated, and the algorithm is then applied repeatedly and recursively for each field of the structure, consuming the available bytes from the byte sequence q.

For *pointer types* (Line 15–21), our current prototype deserializes those as arrays. The value represented by the four bytes consumed from the byte sequence q determine the number of elements (*length*) that are to be included in the array. This includes arrays of *length* one (single elements) and zero (null pointers). This is because in the `C` programming language a pointer can transparently point to one element or the beginning of a list of elements. We can recover the "width" of a single element from the pointee type information ($type'$). The individual elements can then be deserialized recursively. We rely on the user-provided constraints to enforce the validity of the deserialized array length (cf. Tab. I).

Constraint enforcement. In Line 22, the fuzzer runtime modifies the constructed function argument objects to render them valid with respect to the user-provided constraints C. These constraints denote inequalities between constants, primitive type parameters, lengths of array parameters, or array items. In order to "enforce" them, the runtime goes over the deserialized values, bounding the value of each left-hand side of a constraint with respect to the right-hand side.

This implies a dependency relationship between the values of the left-hand side of a constraint and its right-hand side. This in turn means that the set of constraints specified by the user can not denote circular dependencies, in order to allow the runtime to traverse the set of arguments in a valid order.

Additionally, the user can tag string arguments as *filenames*, for which the runtime will dump fuzzing data into a temporary file, and replace the string provided to the function with the corresponding filename.

Serialization. As mentioned earlier, the serialization in Algorithm 1 (Line 8) proceeds analogously, translating the function argument objects o into a byte sequence q, such that if Algorithm 3 is applied to q, we would recover o. However, it might not be immediately clear how pointers are handled. How do we know a priori whether a pointer points to no element at all, a single element, or a number of elements? In this case, we rely on user-specified constraints to properly indicate the size of the array by way of reference. If the user-provided constraints are not strong enough to assign a value to the length of the array referred to by a pointer, our fuzzer prototype defaults to treating character pointers as null-terminated strings (in which case the length of the array is calculated by looking for the first occurrence of the null byte in the string) and any other pointers as pointers to single elements.

Subject	Type	#LOC	Version	Host	AP	M.#C.
boringssl	Encryption	483.2k	dd52194	crypto_test	37	2
bzip2	Compression	8.2k	1.0.8	bzip2	1	2
libass	Rendering	35.4k	0.17.1	ffmpeg	4	2
libexif	Parsing	30.7k	0.6.24	photographer	2	1

TABLE II: Detailed information about our subject programs.

III. Is Execution Amplification Effective?

To study the effectiveness of our in-vivo approach, we implement AFLLIVE and measure the difference in code coverage achieved between the unamplified original executions and the amplified, shadow executions on four target libraries. For each one of them, we choose a host program and one host input to generate unamplified executions, and use a simple CodeQL script to choose interesting amplifier points heuristically.

A. Experimental Setup

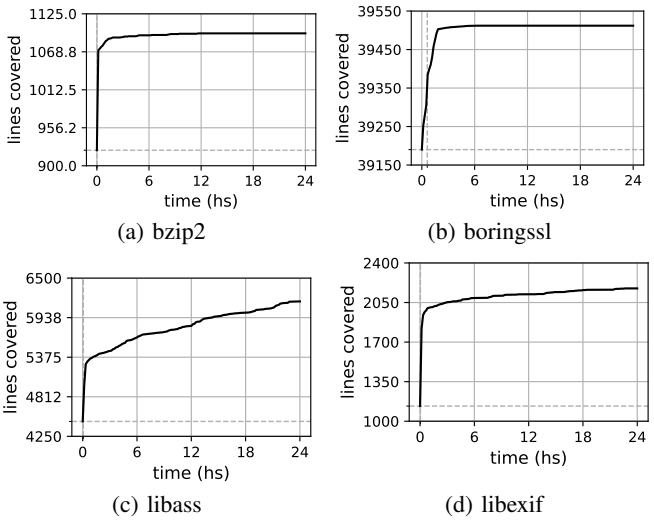
Libraries and hosts. Table II shows information about the benchmarks selected for *this* experiment (two more to follow). We randomly picked four widely-used open-source C libraries that parse host-provided input data. These libraries cover a wide range of domains, from cryptography to rendering. Applications using these libraries might attempt to parse untrusted data and thus any errors present in them might represent potential security vulnerabilities. For every library, we picked one host and one input for that host whose execution we sought to amplify. For our host selection criteria, we focused on programs that were either developed or endorsed by the same group that developed the library. This was done with the intention of minimizing the likelihood of potential crashes stemming from wrong library usage, instead of actual bugs in the library.

For `boringssl`, we used a binary as host that is supposed to test the encryption functionality of the library which also generates the one unamplified execution. For `bzip2`, we used the example application bundled with the source code as host and a compressed version of a text file containing sample text[6] to generate the unamplified execution. For `libass`, we used `ffmpeg` as host, a large video and audio editing library that integrates subtitle functionality via `libass`. The unamplified execution was generated by adding subtitle track with a single subtitle to the shortest possible video. For `libexif`, we used an example application bundled with the source code and one of the test images[7] to generate the unamplified execution.

Amplifier points and constraints are identified using a CodeQL script implementing heuristics to identify parsing-related functions (Section II-A). This script returns potential amplifier points consisting of at most a few hundred functions. We then went through the list, adjusting the automatically inferred constraints based on the function signature and example invocations within the code. Column **AP** in Table II shows how many of the identified amplifier points were executed during the host execution used for the fuzzing campaign. Column **M.#C.** in turn shows the *median* number of constraints specified for each amplifier point, as a "proxy" metric of the amount of effort involved in setting up each subject.

[6]"`The quick brown fox jumps over the lazy dog`"

[7]`22-canon_tags.jpg`



(a) bzip2 (b) boringssl

(c) libass (d) libexif

Fig. 2: Coverage-vs-time for test subjects using in-vivo fuzzing. The horizontal, dashed lines indicate the *baseline coverage* from the original, unamplified execution. The vertical, dashed lines indicate *when* AFLLIVE switched from the screening loop to the main fuzzing loop, on the average.

Fuzzing campaigns. For every project, we started 20 in-vivo campaigns initialized with the same original execution using AFLLIVE. All campaigns were run for 24-hours each on a AMD EPYC 7713P 64-Core processor with 256GB of RAM.

B. Experimental Results

Presentation. The results are shown in Figure 2. All values reflect *coverage within the library*, and not on the host. The vertical, dashed line indicates when, on the average, AFLLIVE switched from screening to the main fuzzing loop. However, since the screening loop only lasts one minute for every target function executed, it is barely visible in cases where few functions were amplified. The horizontal, dashed line indicates the coverage achieved by the unamplified host execution.

Results. The greatest increases in term of coverage were obtained in `libass` and `libexif`, where AFLLIVE achieved an increase of 38.24% (1706 lines) and 91.79% (1040 lines) over the baseline, unamplified execution, respectively. For `bzip2` and `boringssl`, AFLLIVE managed to achieve an increase in coverage of 18.75% (173 lines) and 0.82% (321 lines), respectively.

The lack of increase in coverage for `boringssl` is further confirmed by a visual inspection of Figure 2b where we can see the campaign reach a plateau about 1 hour into the 24 hour fuzzing campaign.[8] Although line coverage did not increase after the first few hours, new path-increasing inputs kept being added to the corpus throughout the campaign. We attribute this

[8]Notice that coverage increases well after the screening loop has finished.

to the fact that out of the 2044 functions executed by the host, 1381 were fully covered in terms of lines by the unamplified host execution. This in turn means there was little room for improvement upon initial line coverage.

The substantial increase in coverage for `libass` and `libexif` is observed over the entire day-long fuzzing campaign and appears to further increase beyond our time budget. This suggests that amplifying the right executions can bring tremendous benefits to automatic vulnerability discovery where the amplified executions reach deep into the code base.

False positives. No crashes were reported in these (previously well-fuzzed) programs during the campaign. This in turn implies that no false positives were reported either, although it is important to keep in mind that the rate of false positives depends on the quality of the specified amplifier constraints.

IV. Onboarding Libraries Without Fuzz Drivers

An advantage of in-vivo fuzzing, as shown in the previous section, is that it makes fuzz drivers superfluous. A *fuzz driver* is a piece of code that acts as a glue code between an off-the-shelf fuzzer and the library-under-test. The driver sets up an artificial calling context and is responsible for accepting data from the fuzzer and feeding it into the library with the appropriate format.

However, effective fuzz drivers are often manually implemented which constitutes a major hinderance to widespread adoption of fuzzing. For instance, Google incentivices the development of fuzz drivers for important open source projects by *paying up to 30k USD* for the successful and effective integration [2]. In fact, Google's project OSS-Fuzz [1] is primarily a community-maintained collection of fuzz drivers for open source projects.

An existing approach to overcome this hindrance is to automatically synthesize fuzz drivers. For instance, FuzzGen [4] leverages a whole system analysis to infer the library's interface and synthesizes fuzz drivers specifically against that interface. Fudge [5] scans a repository for usages of the library's API, uses program slicing [21] to extract the corresponding code snippets, synthesizes a fuzz driver candidate for every code snippet by concretizing place holders, and evaluate the generated fuzz driver candidates by building and running it. IntelliGen [22] also first infers the library's interface annotated with vulnerability likelhoods and generates fuzz drivers for the entry functions through hierarchical parameter replacement and type inference. Daisy [23] first dynamically observes how a host system calls the library's API, and then synthesizes fuzz drivers that follow a similar object usage pattern via a series of API calls.

However, these approaches hoist the tested library only very artificially resulting in a high false positive and false negative rate. The libraries would never be integrated or used in this way in real applications. Approaches that *immitate* the actual usage as faithfully as possible will still not be as close to fuzzing a library as it is actually used. This is precisely our proposal: We suggest to amplify actual user-generated executions where a library is actually used.

In the following, we compare the effectiveness of automatic fuzz driver generation to in-vivo fuzzing as implemented in AFLLIVE. Like fuzz driver generation techniques, in-vivo fuzzing requires only the source code and little human intervention in the specification of amplifier points and constraints.

A. Experimental Setup

Fuzz driver generators. We selected FuzzGen [4] and Fudge [5] according to the following selection criteria. We consider approaches that target `C` libraries, and that are either themselves publicly available and compilable or the generated drivers are publicly available and compilable. We also considered the following fuzz driver generators, but excluded them for the following reasons. GraphFuzz [24] focuses on object-oriented libraries, and in the case of `C` libraries a complete dataflow specification[9] must be provided, which we do not have available. For Daisy [23], despite substantial effort, we did not succeed in compiling the available fuzz drivers due to missing dependencies. For IntelliGen [22], neither the tool itself nor fuzz drivers generated by it were publicly available.

For the comparison against FuzzGen, since the tool itself was not available, we selected three of the seven libraries, as shown in Table III. Out of the four excluded libraries, three had an API that consisted of a single function that accepted a complex `struct` object which wraps the actual library call and maintains the entire state of the library interaction[10]. The remaining library was excluded because it could not be compiled (or easily fixed). For the three selected libraries, we used the only driver available for `libaom` and libvpx and a random fuzz driver (`codlin`) for `libgsm`.

For the comparison against Fudge, we selected all fuzz drivers mentioned in the paper, except OpenCV, as shown in Table III. OpenCV was excluded since the entire API consisted of C++ rather than C functions. `leptonica` and `htslib` are highly popular libraries used for image processing, and high-throughput sequencing data processing, respectively.

Hosts and Original Execution. For every library, we picked one host and one input for that host whose execution we could amplify (cf. Table III). To be fair, we provided each auto-generated fuzz driver with an initial corpus that generates precisely the same values for the library API as our in-vivo fuzzer. *Our intention is that the tested libraries execute on the same piece of data during the first run* (for instance, they should attempt to decode the same byte-stream in the case of decoders). For our host selection criteria, we focused on programs that were either developed or endorsed by the same group that developed the library. This was done with the intention of minimizing the likelihood of potential crashes stemming from wrong library usage, instead of actual bugs in the library.

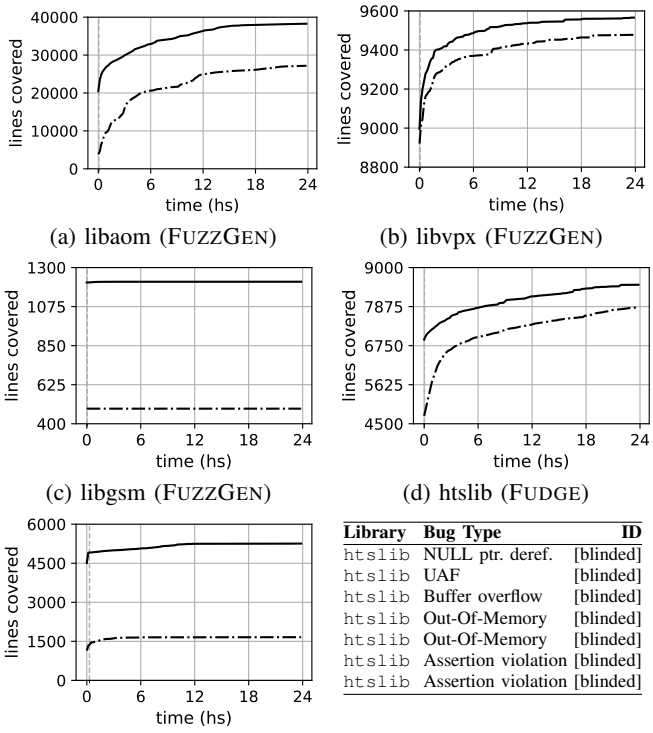
Fuzzing campaigns. For all of the five projects, we started 20 in-vivo fuzzing campaigns initialized with the same original execution using AFLLIVE and 20 normal AFL campaigns using

[9] https://github.com/hgarrereyn/GraphFuzz/issues/1

[10] Example for `libhevc`: https://android.googlesource.com/platform/external/libhevc/+/refs/heads/main/test/decoder/main.c#563

SOTA	Library	Type	#LOC	Version	Synth. fuzz driver	#LOC	Host	Initial corpus	AP	M.#C.
FUZZGEN	libaom	Video Codec	693.0k	3613e5d	av1_dec_fuzzer	1131	aomdec	sample av1 file	4	1.5
	libvpx	Video Codec	0.5k	1.12.0	simple_decoder	482	vpxdec	sample vp9 file	5	2
	libgsm	Speech compressor	8.7k	1.0.22	cod2lin	371	STL/rpedemo	sample wav file	3	1
FUDGE	htslib	File parser	99.0k	1.16	hts_open	152	samtools	sample sam and fasta file	2	2
	leptonica	Image processor	320.0k	1.83.0	pix_rotate_shear	68	tesseract	sample png file with english text	18	1

TABLE III: Summary of setup for each test subjects for comparison against state-of-the-art (SOTA) fuzz driver generators.



(a) libaom (FUZZGEN) (b) libvpx (FUZZGEN)

(c) libgsm (FUZZGEN) (d) htslib (FUDGE)

(e) leptonica (FUDGE)

Library	Bug Type	ID
htslib	NULL ptr. deref.	[blinded]
htslib	UAF	[blinded]
htslib	Buffer overflow	[blinded]
htslib	Out-Of-Memory	[blinded]
htslib	Out-Of-Memory	[blinded]
htslib	Assertion violation	[blinded]
htslib	Assertion violation	[blinded]

(f) Crashes found

Fig. 3: Coverage-vs-time comparison between state-of-the-art (dash-dots) and in-vivo fuzzing (solid), plus crashes found.

the synthesized fuzz harnesses. All campaigns were run for 24-hours each on a AMD EPYC 7713P 64-Core processor with 256GB of RAM.

B. Experimental Results

Presentation. Figure 3 show the results in terms of coverage over time and crashes found for all five subjects. The dashed vertical line indicates when, on the average, AFLLIVE switched from screening to the main fuzzing loop. In all cases, only coverage achieved *within the library* is counted, excluding any coverage information about the host or the fuzz driver.

Coverage results. For the entire duration of the campaign and for all subjects, AFLLIVE achieves substantially more coverage than the campaigns via the synthesized fuzz drivers. Both seem to plateau at around the same time. However, in-vivo fuzzing has the capability to cover substantially more code before reaching that plateau. It is interesting to note that AFLLIVE constistently achieves more initial coverage than AFL via the synthesized fuzz drivers when the campaign is started. We find that a synthesized fuzz driver only exercises a handful of API functions in a rather shallow manner while a host application often interacts with a library via a complex series of API function calls. The synthesized fuzz drivers do not seem to be able to mimic these complex interactions.

The case libgsm seems pathological since there is little coverage increase over time for both fuzzers. Upon closer inspection, we found that the encoding and decoding routines in this library consisted almost entirely of sequential blocks of instructions with no control flow statements. This explains why neither fuzzer was able to increase coverage substantially via the selected amplifier points.

Bug finding results. Our in-vivo fuzzer AFLLIVE found seven previously unknown crashes in htslib (cf. Fig. 3.f). Five were memory safety bugs: a null pointer dereference and two out-of-memory errors within cram/cram_encode.c as well as a heap overflow in header.c, and a use-after-free in md5.c. The remaining two crashes were assertion violations in cram/cram_io.c and cram/cram_codecs.c. AFLLIVE found these seven memory corruption bugs *despite* the FUDGE-synthesized (and later manually adjusted)[11] having continuously fuzzed the library for four years[12]. No further crashes were reported by other campaigns.

False positives. All of the reported crashes were true positives which could be reproduced after the campaigns were finished. Moreover, manual inspection revealed that all seven crashes were reproducible via the host program by providing an appropriate system-level input which, we confirmed, could be under attacker-control.

V. AMPLIFYING THE PROGRAM'S MANUAL TEST SUITE

AFLLIVE can amplify any execution, including one that is generated by a *manually constructed test suite*. Test cases often cover certain edge cases or are designed to catch regressions similar to previously discovered bugs. Over 40% of 0-days exploited in-the-wild are variants of previously discovered vulnerabilities [25]. Amplifying test cases allows us to search their "neighborhood" and bring to light new bugs or those that have been incompletely fixed. Since test suite are designed with code coverage in mind, amplifying test executions might allow us to reach deep into the code, effectively rendering every function ammenable to fuzzing.

[11] https://github.com/samtools/htslib/commits/develop/test/fuzz/hts_open_fuzzer.c
[12] https://github.com/google/oss-fuzz/commit/af319543

Library	Type	#LOC	Version	%Coverage	Initial AP	M.#C.
`openssl`	Cryptography	1M	3.0.6	60%	93	2
`libxml2`	Parsing	308k	2.10.3	61%	14	2
`opus`	Speech compressor	80k	1.3.1	93%	9	2

TABLE IV: Information about libraries and manual test suites.

Distributing energy. Ideally, we would like to fuzz every amplifier point that is executed by the test suite for the same amount of time. However, some amplifier points are executed by a large number of test cases while other amplifier points are executed just by a single test case. So, how much "energy" do we assign to each test case to achieve this objective?

Algorithm 4 Test amplification

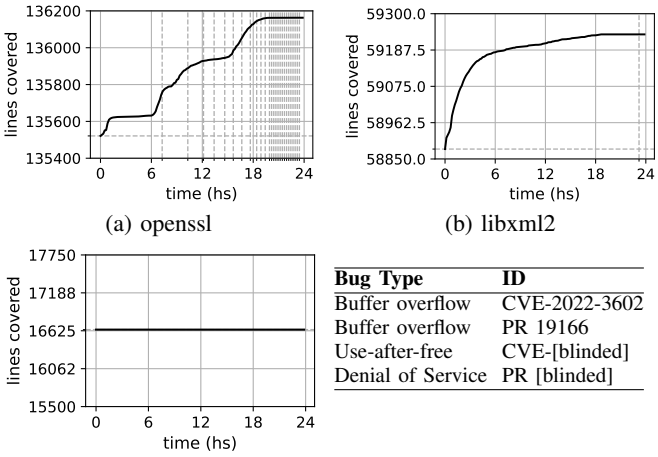
Input: Test suite S
Input: Amplifier points F, Types T, Constraints C, Time t_0
1: Map *test2func* $= \emptyset$
2: Set *funcs* $= \emptyset$
3: **for** Test $s \in S$ **do**
4: *test2func*$[s]$ = `get_exec_amplifiers`(s, F)
5: *funcs* = *funcs* $\cup$ *test2func*[s]
6: **end for**
7: *executed* $= |$*funcs*$|$
8: *fuzzed_funcs*$= \emptyset$
9: **while** not aborted **do**
10: **for** s in S **do**
11: *unfuzzed* $= |$*test2func*$[s] -$ *fuzzed_funcs*$|$
12: **if** *unfuzzed* > 0 **then**
13: Time budget t_1 = *unfuzzed*/*executed*
14: `fuzz`$(F, T, C,$ `exec`$(s), t_0, t_1)$
15: *fuzzed_funcs* = *fuzzed_funcs* $\cup$ *test2func*$[s]$
16: **end if**
17: **end for**
18: **end while**

Algorithm 4 illustrates our algorithm to distribute the available energy evenly over the amplifier points executed by the test suite S. In Line 1–7, it finds the amplifier function executed by each test case $s \in S$ and counts how many amplifiers are executed in total. In Line 8–18, it skips test cases that execute no unfuzzed amplifier point (Line 12). Otherwise, it computes the proportion of all executed amplifiers that are executed by test case s and still unfuzzed as the time budget t_1 for s (Line 13), and starts a corresponding fuzzing campaign (Line 14). Specifically, the function `fuzz` implements the proposed in-vivo fuzzing approach as defined in Algorithm 1.

A. Experimental Setup

Table IVshows the selected libraries, the corresponding test suite coverage, and the number of executed amplifier points (AP). We randomly chose libraries from diverse domains that are security-critical, well-fuzzed (5+ years),[13] and widely used open-source C libraries. For test amplification, no host or host input is needed, as all libraries had test suites and testing frameworks readily available. Like for the other experiments, the amplifier points were auto-identified using our tool (§ II-A) and manually constrained afterwards.

[13]2016 Commit contains OpenSSL & LibXML2: https://github.com/google/oss-fuzz/commit/a143b9b3



(a) openssl (b) libxml2

(c) opus

Bug Type	ID
Buffer overflow	CVE-2022-3602
Buffer overflow	PR 19166
Use-after-free	CVE-[blinded]
Denial of Service	PR [blinded]

(d) Bugs found in OpenSSL.

Fig. 4: Coverage and bugs in test amplification campaigns.

State of the art. There exists a fuzz driver generator specific for test amplification, called UTopia [6]. Given a library-under-test and the `gtest` or `boost` test suite, UTopia first performs a lightweight static analysis before synthesizing fuzz drivers for the tested library functions. The *static analysis* is used to identify the precondition of every library function. For every test case, the *synthesis* first identifies the library functions used in the test case and the constants used as parameters in a corresponding function call, and then generates a fuzz driver for the library functions by rendering the constant library function call parameters subject to fuzzing. For our experiments, we reuse the identified functions and preconditions as amplifier points and constraints using a straightforward translation, to ensure the fairness of the comparison. This demonstrates the versitality of our in-vivo approach which allows diverse means of automatic amplifier point identification and requires no specific test framework.

Unfortunately, despite several months of experimentation, we realized that *on the* UTopia *benchmark programs using the* UTopia*-identified amplifier points and constraints*, all crashing inputs generated by UTopia (and by our AFLLive) *only reveal false positives*. Upon manual examination, we discovered that the drivers synthesized by UTopia (as well as the results of its analysis) did lead to an incorrect usage of the libraries, and thus to a large amount of spurious crashes. To be sure, we repeated the analysis by filtering inputs that did not crash on the most recent version, assuming these bugs would now be fixed, but only found that the remaining crashers were flaky, i.e., crashed again if run repeatedly. Since the prototype provided by the authors is highly automated (i.e. it requires little intervention and there is not much room for misusage), we conclude that an experimental comparison would not provide much insight.

B. Experimental Results

Presentation. Figure 4 shows the average coverage over time and the bugs found during test amplification. The vertical dashed lines indicate a change in test case during the fuzzing campaign (Line 14 of Alg. 4). The horizontal dashed line indicates the initial coverage for the library's test suite. We only measure coverage of the library.

Coverage results. AFLLIVE achieved an increase over the manual test suite by around 600 LoC for `openssl` and over 300 LoC for `libxml2`. No increase in coverage was achieved for `opus`. Closer inspection revealed that the manual test suite is of very high quality and nearly saturated, covering almost 95% of lines of code in opus. There are five test cases that exercise all of the amplifier points selected for opus (which explains why all of the time budget was invested into one test case). The latter is true also for LibXML2, where the first test case already exercises all but one amplifier point.

For `openssl`, we see that switching test cases to exercise new amplifier points is effective and after saturation, code coverage increases again when the next test case is fuzzed. This highlights that, using our approach, once the amplifier points have been identified and their constraints correctly specified the user is able to setup several fuzzing campaigns with little extra effort. Towards the end of the 24 hour campaign, it is also interesting to note that coverage saturates despite switching to test cases that exercise new amplifier points.

Bug finding results. AFLLIVE discovers 4 bugs in `openssl`, two of which have previously been found only by manual auditing, including the *high*-severity PunyCode vulnerability (CVE-2022-3602), and two of which have not previously been known, including a *moderate*-severity use-after-free (CVE-[blinded]). In terms of *false positives*, no false positive crashes were reported.

VI. Semi-Automated Identification of Amplifier Points and Constraints

The two main concepts of invivo fuzzing are *amplifier points* (APs) and *amplifier constraints* (ACs). While APs identify interesting functions, the purpose of ACs is to make implicit function-preconditions explicit, just like like user-defined preconditions in property-based testing (PBT) [18], [19] or user-defined repOK-methods in search-based software testing (SBST) [26], [27].

In general, ACs can be written manually to reduce false positives, but they do not *need* to be added. In terms of manual effort, there is a tradeoff between specifying ACs versus going through the false positives. For instance, suppose AFLLIVE finds a possible null-pointer-dereference on a function-parameter, but that function is never called with a null-pointer. This is a false positive. We allow users of invivo-fuzzing to encode this implicit assumption explicitly.

A. Semi-Automatic Identification

For our experiments, we used a semi-automated approach. An initial set of APs/ACs was first automatically identified and then manually refined. For automation, we developed a CodeQL script to identify APs (113 LoC) and a Python script to generate ACs (274 LoC).[14] For manual refinement:

- For subjects where no executed APs where identified (`bzip2`, `libexif` and `libgsm`), we added the main entry points of the library as APs (via documentation).
- We added (or modified) ACs to ensure these conditions:
  1) (`sizeof(buf)` $=$ *len* $\wedge$ *len* $<$ *C*) which requires that variable len determines the length of the buffer *buf* and *len* is less than the constant C.
  2) (`sizeof(buf)`$<$*C*) which requires that the length of the buffer is smaller than the constant C, or
  3) (`is_file`*(filename)*) which requires that the string $filename$ refers to a valid file where fuzzing input will be dumped.

These patterns account for 98% of all ACs.

As an indicator of the additional manual effort for each subject, we note that we either added or removed constraints for no more than 12% of the automatically identified amplifier points across all subjects. Even then, no more than two constraints needed to be added/removed.

In comparison to AC specification, writing a fuzz driver from scratch could take an experienced developer several hours, and would need to be maintained afterwards. For instance, the driver integrated into OSS-Fuzz [1] for `libass` was written over the course of two days by a core developer of the project, and iterated upon several times.

Subject	Inferred config. Cov. (#LOC)	T.P.	F.P.	Curated config. Cov. (#LOC)	T.P.	F.P.
`boringssl`	40027.95	-	4	39511.95	-	-
`bzip2`	-	-	-	1096.10	-	-
`libass`	6553.40	-	-	6168.25	-	-
`libexif`	-	-	-	2174.95	-	-
`htslib`	5723.00	-	-	8504.50	7	-
`leptonica`	3178.85	-	-	5255.75	-	-
`libaom`	11442.00	-	-	38261.55	-	-
`libgsm`	-	-	-	1218.40	-	-
`libvpx`	8203.00	-	-	9565.80	-	-
`libxml2`	59681.40	-	-	59235.95	-	-
`openssl`	135903.20	-	4	136162.90	4	-
`opus`	18804.95	-	-	16642.00	-	-

TABLE V: Coverage and bugs in fully automated campaigns.

B. Ablation Study

In order to study the impact of our additional manual effort to reduce false positives, we compare the effectiveness of AFLLIVE using only the auto-generated APs and ACs to the effectiveness of AFLLIVE using the manually augmented set of APs and ACs. All campaigns were run for 24-hours each on a AMD EPYC 7713P 64-Core processor with 256GB of RAM.

Coverage results. Table V shows the average coverage achieved throughout the campaign, along with false and true positives reported, for both the fully automated and manually modified configurations. For all subjects with identifiable amplifier points, coverage achieved via auto-generated APs

[14] https://anonymous.4open.science/r/afllive-598A/config_generator

and ACs was on par (i.e., same order of magnitude) with the coverage achieved through the semi-automatic approach.

For the subject where no executed APs were identified automatically (i.e., `bzip2`, `libexif` and `libgsm`), the campaigns failed to run. However, after manually specifying 6 amplifier points and 7 constraints across the three subjects, the campaigns run and managed to increase coverage significantly over the original execution (see Figure 2, Figure 3).

For some subjects the automatically inferred constraints led to a higher code coverage, such as in the case of `boringssl`, `libass`, `libxml2` and `opus`. This can be attributed to the fact that we were overly conservative when manually modifying constraints in an effort to prevent a high false-positive rate.

Bug finding results. Given only the automatically inferred constraints, AFLLIVE failed to find the previously discovered bugs. Expectedly, this also led to a higher number of false positives for two of the subjects (`boringssl` and `openssl`), which were also the most complex subjects that we analyzed. Still, no more than five false positives were reported in each case, and could thus be triaged in a reasonable amount of time (less than a few hours).

VII. Related Work

Automatic unit-level testing. Long before fuzzing entered the stage, the software engineering community studied automatic approaches for unit-level test generation [18], [26], [28]–[30]. Examples of a *unit* are Java objects or C functions. One *major research challenge* of automatic unit-level testing has been to minimize the number of *false positives*, i.e., bugs that only appear during automatic testing, but never in production when the unit is properly used. There are two approaches to tackle this problem: (a) to let the user specify conditions representing the valid usage of that unit [18], [26], and (b) to observe how the unit is used, e.g., during system-level testing, and to enforce the infered protocol during unit-level testing [31]. For instance, the approach taken by the Daisy [23] fuzz driver generator represents Approach (b) while our AFLLIVE takes Approach (a) to minimize the number of false positives during in-vivo fuzzing.

Valid calling context. Another major research challenge of automatic unit-level testing has been to generate a valid sequence of API calls and construct the required objects to pass in as parameters to these calls. Given the preconditions (called contract), Randoop [29] constructs the sequence of API calls and objects in a feedback-directed manner, continuously evolving test cases that do not violate the user-provided contract. JQF [19] and CGPT [32] add coverage-guidance. However, fundamentally these tools follow a generational approach where the API calls and objects are generated out of thin air and validated only against a user-provided specification. In contrast, ours is a mutational approach, where we *piggyback* on a valid sequence of API calls that are passed valid objects. Like the mutational approach on the system-level [3], [7], [20], this allows us to reach much deeper into the code. Staying within the neighborhood of a valid program state, there is a low risk of false positives.

In-vivo fuzzing in production. Our long-term vision, assuming several technical challenges are tackled, is to integrate in-vivo fuzzing into the production system, so as to fuzz the entire supply chain of a software system, including all of its dependencies. The idea to integrate bug finding into production is not very far fetched. For instance, Google is running a no-overhead version of AddressSanitizer [33] on every Android 11 phone and every Chrome browser [34], [35]. Apart from bug finding, Google has long been running Google-Wide Profilers (GWP) which conduct light-weight program analysis across entire fleets of machines [36]. Mozilla implemented the approach for Firefox [37]. The open source community implemented the approach for the Linux kernel [38].

VIII. Conclusion

A. Perspective

Existing fuzzers are designed to test a software system *in-vitro*, i.e. under artificial lab conditions. However, the effectiveness of in-vitro fuzzing is limited [39]. It is these limitations which we sought to address in this paper.

Solving dependency on fuzz driver quality. A fuzzer must first be "glued" to the software via fuzz drivers. Typically, fuzz drivers are tediously developed and continuously updated over months. For instance, Google pays up to 20k USD for fuzz drivers of critical open source software [1], [40]. To reduce some manual effort, recent research has focussed on generating drivers automatically [4], [5], [41]. Whenever a security was found by manual auditing, the developer would add a new fuzz driver through which the fuzzer is able to find the security flaw. While the drivers can be improved over time, this dependency on driver quality cannot be avoided. OpenSSL has 16 drivers in OSS-Fuzz which have been continuously fuzzed 24/7 over the past six (6) years [42]. *In contrast*, in-vivo fuzzing eliminates the need for fuzz drivers entirely. Just by amplifying the developer test suite, our in-vivo prototype found a critical bug in "unfuzzed" code of OpenSSL (CVE-2023-0215).

Solving structure-aware fuzzing. A fuzzer's effectiveness depends critically on the quality of the initial seed corpus [43]. For instance, if we are fuzzing an PNG image library, inputs that were generated by mutating valid PNG image files will reach more deeply into the library than a random string of bytes. However, valid input structures are easily broken and new input structures are difficult to generate by chance. For instance, if none of the seed images contains an optional `eXIf` chunk specifying some metadata, it will hardly be generated. Recent work, including ours [8], has addressed this using (or learning) the input structure, and "inventing" the missing data chunks [44]–[47]. However, the critical dependence on initial seeds remains. *In contrast*, in-vivo fuzzing allows us to define as amplifier point that function in the parser which handles an interesting data chunk or set amplifier points deep in the program functionality to entirely skip the parser.

Solving stateful fuzzing. Some software systems require inputs in a certain order. For instance, the Transmission Con-

trol Protocol (TCP) requires a three-way handshake between client and server before data can actually be sent. Without knowing precisely the implemented protocol, it is difficult for a fuzzer to generate the right sequence of packets with the correct structure. Recent work, including ours [12], has used mutational, feedback-direct fuzzing that uses response codes, state variables, or human annotations to identify and leverage the sequence of software states for a sequence of inputs/packets [14], [48]. However, these approaches heavily depend on the recorded sequences of packets that are used to seed the mutational fuzzers. *In contrast*, in-vivo fuzzing allows us to define as amplifier point that function which handles a certain state or state transition.

B. Paper Summary

Our approach allows the user to fuzz a library within the context of a host application by exploring the neighborhood of a valid program state induced by an actual host-generated execution of that library. We do so by applying coverage-guided mutation-based fuzzing on the arguments of each function marked as an amplifier point, subject to a set of user-specified constraints. By using real-world programs, we can leverage our approach to fuzz the library within a production-like usage context. Conversely, we can use a test-suite as a host to explore variants of regression tests and corner cases identified by the developers. In contrast to a fuzz-driver based approach, selected amplifier points need not be part of the API, implying that our approach can reach deeper into the code.

In our experiments we manage to increase coverage significantly over non-amplified executions, indicating that amplification is indeed effective. Furthermore, we manage to outperform existing state-of-the-art approaches for automated fuzz-driver generation in terms of both code coverage and bug finding. Providing empirical evidence is the discovery of seven (7) previously unknown vulnerabilities in `htslib`, even as this library has been continually fuzzed using synthetic fuzz drivers for seven (7) years as of the time of writing. This not only suggests that execution amplification is effective, but also that real-world applications do indeed interact with libraries in ways that are not properly captured by existing fuzz drivers. Moreover, through test amplification we re-discover a high-severity vulnerability in `openssl` and also uncover a novel moderate severity vulnerability, both of which had not been found through fuzzing before. Apart from the vulnerabilities, we find a known bug and a novel one, as well.

We should note that the effectiveness of our approach depends crucially on the choice of amplifier points and constraints. If we choose the wrong amplifiers, we might get false positives crashes; but given the flexibility of our approach, we did not find this to be an obstacle. For our experiments, we developed a CodeQL script to come up with an initial amplifier set.[15] Via an interactive process, we refined the constraints (i.e., preconditions) for every function as follows: Whenever a constraint was incorrectly specified, the fuzzer would fail within a few seconds, and the constraint would need an obvious adjustment. Overall, the amplifier identification process took no more than a few hours for every library.

There are still several interesting socio-technical challenges ahead of us. Considering that the largest continuous fuzzing platform, OSS-Fuzz [1], which fuzzes over 1000 open source projects on 100k machines 24/7, is nothing but a collection of manually generated fuzz drivers, we are truely excited about the prospect that in-vivo fuzzing enables fuzzing for every library that is used and compiled in a production environment.

[15] https://anonymous.4open.science/r/afllive-598A/README.md#config-file-1

References

[1] K. Serebryany, "OSS-Fuzz - google's continuous fuzzing service for open source software," in *USENIX Security*. Vancouver, BC: USENIX Association, Aug. 2017.

[2] OSS-Fuzz, "Integration rewards," https://google.github.io/oss-fuzz/getting-started/integration-rewards/, 2021, accessed: 2023-01-11.

[3] LLVM, "Libfuzzer," https://llvm.org/docs/LibFuzzer.html, accessed: 2023-01-11.

[4] 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

[5] 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*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 975–985. [Online]. Available: https://doi.org/10.1145/3338906.3340456

[6] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2676–2692.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[8] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2021.

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

[10] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[11] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 244–256. [Online]. Available: https://doi.org/10.1145/3460319.3464814

[12] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[13] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 337–350. [Online]. Available: https://doi.org/10.1145/3460120.3484543

[14] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 166–180. [Online]. Available: https://doi.org/10.1145/3492321.3519591

[15] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 330–347.

[16] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[17] Github, "Codeql," https://codeql.github.com/, 2021, accessed: 2023-01-11.

[18] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 268–279. [Online]. Available: https://doi.org/10.1145/351240.351266

[19] R. Padhye, C. Lemieux, and K. Sen, "Jqf: Coverage-guided property-based testing in java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 398–401. [Online]. Available: https://doi.org/10.1145/3293882.3339002

[20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[21] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. IEEE Press, 1981, p. 439–449.

[22] 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)*, 2021, pp. 318–327.

[23] M. Zhang, C. Zhou, J. Liu, M. Wang, J. Liang, J. Zhu, and Y. Jiang, "Daisy: Effective fuzz driver synthesis with object usage sequence analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2023, pp. 87–98.

[24] H. Green and T. Avgerinos, "Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1070–1081. [Online]. Available: https://doi.org/10.1145/3510003.3510228

[25] M. Stone, "The ups and downs of 0-days: Our review of 0-days exploited in-the-wild in 2022," July 2023, accessed: 2023-01-11.

[26] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 123–133. [Online]. Available: https://doi.org/10.1145/566172.566191

[27] D. Marinov and S. Khurshid, "Testera: a novel framework for automated testing of java programs," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001, pp. 22–31.

[28] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *International Conference On Quality Software (QSIC)*. IEEE Computer Society, 2011, pp. 31–40.

[29] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 75–84.

[30] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294

[31] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 253–264. [Online]. Available: https://doi.org/10.1145/1181775.1181806

[32] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360607

[33] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[34] M. Morehouse, M. Phillips, and K. Serebryany, "Crowdsourced bug detection in production: Gwp-asan and beyond," in *Proceedings of the C++ Russia*, 2020.

[35] V. Tsyrklevich, "Gwp-asan: Sampling heap memory error detection in-the-wild," https://www.chromium.org/Home/chromium-security/articles/gwp-asan, accessed: 2023-01-11.

[36] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, pp. 65–79, 2010. [Online]. Available: http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68

[37] C. Holler, "Phc (probabilistic heap checker): a port of chromium's gwp-asan project to firefox," https://bugzilla.mozilla.org/show_bug.cgi?id=1523268, 2021, accessed: 2023-01-11.

[38] L. K. Developers, "Kernel electric-fence (kfence)," https://www.kernel.org/doc/html/latest/dev-tools/kfence.html, 2021, accessed: 2023-01-11.

[39] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021.

[40] O.-F. Team, "Oss-fuzz integration awards," https://google.github.io/oss-fuzz/getting-started/integration-rewards/, accessed: 2023-01-11.

[41] 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

[42] "Openssl at oss-fuzz: Commit history," https://github.com/google/oss-fuzz/commits/master/projects/openssl, accessed: 2023-01-11.

[43] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 230–243. [Online]. Available: https://doi.org/10.1145/3460319.3464795

[44] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "Slf: Fuzzing without valid seed inputs," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 712–723. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00080

[45] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 627–637. [Online]. Available: https://doi.org/10.1145/3106237.3106295

[46] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[47] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3395363.3397372

[48] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy*, ser. S&P 2020, 2020, pp. 1597–1612.

[49] P. Godefroid, "Micro execution," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 539–549. [Online]. Available: https://doi.org/10.1145/2568225.2568273

[50] W. Gao, V.-T. Pham, D. Liu, O. Chang, T. Murray, and B. I. Rubinstein, "Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report)," in *Proceedings of the 2nd International Fuzzing Workshop*, ser. FUZZING 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 47–55. [Online]. Available: https://doi.org/10.1145/3605157.3605177

[51] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012,

pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler
[52] T. Dullien, “Introducing prodfile,” https://prodfiler.com/blog/, 2021, accessed: 2023-01-11.
[53] Google, “Syzkaller: an unsupervised coverage-guided kernel fuzzer,” https://github.com/google/syzkaller, 2021, accessed: 2023-01-11.