Liberating Libraries through Automated Fuzz Driver Generation: Striking a Balance without Consumer Code

FLAVIO TOFFALINI, Ruhr University Bochum, Germany and EPFL, Switzerland
NICOLAS BADOUX, EPFL, Switzerland
ZURAB TSINADZE, EPFL, Switzerland
MATHIAS PAYER, EPFL, Switzerland

Fuzz testing a software library requires developers to write *fuzz drivers*, specialized programs exercising the library. Given a driver, fuzzers generate interesting inputs that trigger the library's bugs. Writing fuzz drivers manually is a cumbersome process and they frequently hit a coverage plateau, calling for more diverse drivers. To alleviate the need for human expert knowledge, emerging automatic driver generation techniques invest computational time for tasks besides input generation. Therefore, to maximize the number of bugs found, it is crucial to carefully balance the available computational resources between generating *valid drivers* and testing them thoroughly. Current works model driver generation and testing as a single problem, *i.e.,* they mutate both the driver's code and input together. This simple approach is limited, as many libraries need a combination of non-trivial library usage and complex inputs. For example, consider a JPEG manipulation library, bugs appear when specific library functions and corrupted images are coincidentally tested together, which, if both are mutated synchronously is difficult to trigger.

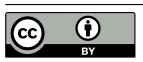
We introduce libErator, a novel library testing approach that balances constrained computational resources to achieve two goals: (a) quickly generate valid fuzz drivers and (b) deeply test these drivers to find bugs. To achieve these goals, libErator employs three main techniques. First, we leverage insights from a novel static analysis on the library code to improve the likelihood of generating meaningful drivers. Second, we design a method to quickly discard non-functional drivers, reducing even further resources wasted on unfruitful drivers. Finally, we show an effective driver selection method that avoids redundant tests. We deploy libErator on 15 open-source libraries and evaluate it against manually written and automatically generated drivers. We show that libErator reaches comparable coverage to manually written drivers and, on average, exceeds coverage from existing automated driver generation techniques. More importantly, libErator automatically finds 24 confirmed bugs, 21 of which are already fixed and upstreamed. Among the bugs found, one was assigned a CVE while others contributed to the project test suites, thus showcasing the ability of libErator to create valid library usages. Finally, libErator achieves 25% true positive ratio, doubling the state of the art.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Fuzzing, Library Testing, Driver Generation.

ACM Reference Format:
Flavio Toffalini, Nicolas Badoux, Zurab Tsinadze, and Mathias Payer. 2025. Liberating Libraries through Automated Fuzz Driver Generation: Striking a Balance without Consumer Code. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE095 (July 2025), 23 pages. https://doi.org/10.1145/3729365

Authors' Contact Information: Flavio Toffalini, Ruhr University Bochum, Bochum, Germany and EPFL, Lausanne, Switzerland, flavio.toffalini@rub.de; Nicolas Badoux, EPFL, Lausanne, Switzerland, nicolas.badoux@epfl.ch; Zurab Tsinadze, EPFL, Lausanne, Switzerland, zurab.tsinadze@epfl.ch; Mathias Payer, EPFL, Lausanne, Switzerland, mathias.payer@epfl.ch.



ACM 2994-970X/2025/7-ARTFSE095
https://doi.org/10.1145/3729365

1 Introduction

Fuzzing has unparalleled bug-finding capabilities [9, 15, 48, 54]. This automated testing technique stochastically samples a program's input space, often thousands of times per second, to trigger flaws. General-purpose fuzzing engines expect a well-defined interface (*e.g.,* a `main` function) to run the code under test. This simple but effective design paved the way for a variety of generic and specialized fuzzers [3, 4, 6, 9, 15, 27, 31, 43, 44, 48]. Unfortunately, not all targets have fuzzing-compatible interfaces. In particular, user-space libraries (*e.g.,* `libpng` [34]) are designed to be integrated into stand-alone programs. Specifically, libraries expose a set of functions, the Application Programming Interface (API), to interact with the library code.

To tailor a fuzzer workflow to libraries, practitioners write small snippets of code (*i.e., drivers* or *fuzz drivers*) to tie the fuzzing engines to the library code. While some initiatives reduce the cost of fuzzing library drivers—*e.g.,* OSS-Fuzz [37], Google's effort to continuously test open-source libraries—driver creation remains, however, predominantly a manual effort [37]. Due to the required knowledge of the library's API and the scarcity of maintainers time, manually-written drivers are limited in the extent of their fuzzing campaigns as well as their capacity to evolve with the library changes. As we observe in OSS-Fuzz, fuzzed projects have generally reached a coverage plateau where the existing drivers no longer discover new functionalities. To overcome this limitation, academia and industry investigated approaches to create drivers automatically—trading maintainers' time for CPU resources—with the goal of covering new untested code paths and thereby exposing bugs in the targets [1, 5, 18, 22, 23, 47, 50, 51, 53].

The first explored approach to generate fuzz drivers is *consumer-dependent*, which analyzes the interaction of existing applications (*consumers*) with a library [1, 22, 23, 47, 50, 51, 53]. *Consumer-dependent* drivers are bounded to the patterns found in the consumers analyzed. To overcome this limitation, *consumer-agnostic* techniques [5, 18] rely solely on the library code and apply static or dynamic techniques to infer valid library usages, thus finding bugs that may not appear in the known consumers. All current *consumer-agnostic* approaches, and some *consumer-dependent* works [22, 51], attempt to infer *valid library usage* ***and*** *valid inputs simultaneously*, thus solving two orthogonal problems at once, leading to a suboptimal solution due to the exponential growth of the input space. In practice, however, the computational budget is a finite resource T used to solve two distinct tasks: generate drivers (with the goal of maximizing potentially reachable coverage) and explore drivers through inputs (with the goal of maximizing concrete coverage and bug finding). In other terms, the time for driver generation (t_{gen}) and testing (t_{test}) is bounded by T, *i.e.,* $t_{\text{gen}} + t_{\text{test}} = T$. Manually written drivers allocate only testing time, while automatic driver generation mechanisms split testing and generation according to different policies. In Figure 1, we classify previous works according to their time budget allocation strategy.

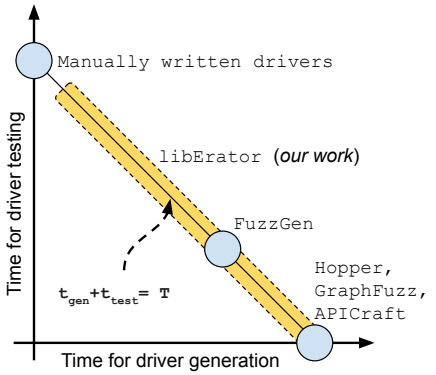


Fig. 1. The driver generation t_{gen} and the driver testing t_{test} sum up to the total computational time T, *e.g.,* $T = t_{\text{gen}} + t_{\text{test}}$. LIBERATOR (in yellow) can be configured with any balance of t_{gen} and t_{test} to fit the library.

LIBERATOR introduces a library testing model that balances resources between generation and fuzzing by leveraging three main techniques. First, *API sequence inference*: through static analysis, we infer which API sequences are more likely to contribute to coverage, allowing for fast and promising drivers. Second, through *dynamic pruning of ineffective sequences*, LIBERATOR avoids

wasting time on unfruitful drivers. We propose to promptly discard broken API function sequences and avoid extending them further. Finally, since the aim is to test different code regions inside a library, we propose to *balance driver diversity to optimally distribute fuzzing energy* through a lightweight driver selection strategy that diversifies the API functions used.

We evaluate libErator by targeting 15 libraries representing varying API and input space complexity. As a baseline, we employ existing state-of-the-art solutions. We compare our approach against manually written drivers from the OSS-Fuzz project [37], *consumer-agnostic* works, Hopper [5], and *consumer-dependent* works, namely UTopia [23], FuzzGen [22], and OSS-Fuzz-Gen [21]. Comparison with OSS-Fuzz demonstrates that libErator's drivers perform deep and complex library interactions, similar to the ones written by experts, *i.e.,* for six out of 12 libraries we achieve higher coverage. While, compared with Hopper, we reach more coverage in eight out of the 13 libraries supported. Against consumer-dependent approaches, we achieve comparable coverage for UTopia despite their spurious use of internal API functions, and exceeds FuzzGen and OSS-Fuzz-Gen. Most importantly, we found 24 confirmed bugs in libraries already extensively tested by OSS-Fuzz, Hopper, or OSS-Fuzz-Gen. We upstream fixes for 21 of them and are in discussion with maintainers to contribute the respective drivers. Moreover, libErator experiences 25% of true positives, which doubles similar works [23]. These experiments demonstrate the capability of libErator to improve library testing capabilities.

In short, our key contributions are:

- A *consumer-agnostic* library model that balances driver generation and testing. Our model automatically generates valid library interactions and discovers code faults.
- We build our library model on three techniques: *API sequence inference* to proactively hint at promising sequences of API function calls, *dynamic pruning of ineffective sequences* to learn dysfunctional sequences, and *driver selection technique* to better distribute testing energy.
- libErator, an end-to-end framework that implements our *consumer-agnostic* library model and generates fuzz drivers for libraries by solely relying on their source code. We release the implementation of our prototype as open source, see §11.
- A detailed evaluation of our results against the state-of-the-art driver generation technique and manually written drivers, discussing the trade-offs of each approach.

2 Automatic Library Testing

Fuzz testing or fuzzing is a dynamic testing approach that leverages high execution throughput to sample the input space and discover bugs. Guided fuzzers [9, 48] leverage execution feedback to bias the input generation towards bug-prone code paths. These techniques have shown their effectiveness in industry [13, 14, 49] and are widely deployed, *e.g.,* thousands of projects are continuously fuzzed by OSS-Fuzz [37]. However, not all software is equally amenable to fuzzing.

Software libraries provide functionalities that, otherwise, should be re-implemented in each project with the risk of repeating past mistakes and bugs. Interactions between the main program and a library happens through functions part of the Application Programming Interface (API). To enable a fuzzer to test a library, an entry point, taking the form of a program, has to be created. These programs, called drivers, might require non-trivial inputs (*e.g.,* files) and should build the necessary state to interact, ideally, with the full library's API. libFuzzer [35], the pioneer in library fuzzing, models drivers as stub functions, called `LLVMFuzzerTestOneInput`, which takes as input a buffer of bytes. This flexible interface has been adopted as a standard in other popular fuzzing tools [9, 48]. Notably, libFuzzer executes drivers in a loop with different inputs to minimize the startup overhead. Therefore, drivers must exit cleanly, *e.g.,* by invoking `free` or closing files, to avoid lingering states leading to non-reproducible bugs. The number of drivers remains limited as they are manually written, *e.g.,* only *eight* out of the 15 libraries evaluated have multiple drivers.

```
1  extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
2   vpx_codec_ctx_t codec;
3   vpx_codec_dec_cfg_t cfg;
4   size_t frame_size;
5   uint8_t *frame; /* size(frame) == frame_size */                    ①
6    ...
7   cfg = ... /* from data */
8   frame_size = ... /* from data */
9   frame = malloc(frame_size);
10  memcpy(frame, /* from data */, frame_size);                        ②
11  /* codec cannot be populated from data! */
12   ...
13  vpx_codec_dec_init(&codec, VPXD_INTERFACE(DECODER), &cfg, 0);
14  const vpx_codec_err_t err = vpx_codec_decode( &codec,frame, frame_size, nullptr, 0);   ③
15   ...
16  vpx_codec_destroy(&codec);                                        ④
17  return 0;
18 }
```

Listing 1. Simplified example of a C driver from `libvpx` highlighting the four driver regions: ① variable definition, ② input transfer, ③ API chain execution, and ④ state cleanup.

Listing 1 presents an abbreviated driver written by the `libvpx` maintainers [17]. Without loss of generality, we decompose the driver in *four* regions: ① variables definition, ② input transfer, ③ API chain execution, and ④ cleanup. Region ① declares the necessary variables for the driver. Region ② bridges the fuzzing input into the respective variables. However, not all the variables can be populated with raw data, *e.g.,* `codec` has internal pointers fields and requires invocation of `vpx_codec_dec_init`. Region ③ consists of the *API chain*, *i.e.,* a valid sequence of API calls. *API chains* are composed of a valid sequence of function calls and correct variables as their respective arguments. Finally, region ④ cleans the driver state to avoid memory leaks, *e.g.,* `vpx_codec_destroy` releases `codec` at Line 16.

3 Challenges for Automatic Driver Generation

Driver generators aim at inferring valid and interesting API Chains (§2). However, not all API function call combinations are valid. Producing *valid* chains introduces challenges exemplified through a combinatory exercise. Given a library that exposes N API functions, an under-approximation of the number of possible API Chain is given by $N^{|API\_Chain|}$. This means that the search space grows exponentially with the chain size. This approximation is conservative as it does not consider function arguments. On top of that, each API Chain has its own input space that can be modeled as a bitstream of length I. On classic targets, the fuzzers only sample the input space (*i.e.,* 2^I). Conversely, in automatic library testing, the system is also responsible for generating drivers, thus extending the search space to at least $N^{len(API\_Chain)} \times 2^I$. Therefore, creating valid API Chains is an additional dimension requiring ad-hoc reasoning. We study how current approaches model library testing, and propose a new solution to better address this extra dimension.

Current works generate drivers by synthesizing API Chains *and* searching for valid inputs simultaneously [5, 18, 22, 51]. They assume coverage to be a fair feedback, *i.e.,* API Chains reaching more coverage should be expanded, leading to critical limitations. First, API Chains may face a coverage wall, meaning they do not reach new coverage, however, the fuzzer needs more time to pierce through the perceived coverage wall. When this occurs, current works misjudge the API

Chain as unpromising and stop testing it. Second, libraries may require a long sequence of API function calls to initialize complex structures to unlock access to deep library code. However, prefix API Chains may not reach meaningful coverage and thus be ignored, despite being crucial for longer API Chains. Third, current works produce new API Chains continuously, without testing them deeply. Hence, an explosion of undertested API Chains hinders the overall testing progress.

By surveying the state-of-the-art, we define three challenges that drive libErator design.

(C1) Predicting complex API combinations. To reduce the computational cost associated with the generation, it is crucial to predict which API Chains are interesting.

(C2) Avoiding coverage wall biases. Come up with a strategy to learn, and avoid building on, unfruitful API Chains that should not suffer from biases created by coverage walls.

(C3) Deep testing of API Chains. To efficiently fuzz, libErator needs to avoid redundant testing of similar API Chains and invest computation cycles on diverse library usages.

4 Driver Specification

In this section, we detail the characteristics of the generated drivers. libErator's prototype creates functional and valid drivers written in C, compatible with existing fuzzers like libFuzzer [35] or AFL++ [9]. Therefore, libErator needs to use coherently the variables passed to the API functions. This constrains some technical choices, such as the type system and the handling of the variables lifetime. Most of our solutions extends ideas from previous works [5, 22]. libErator's design is language-agnostic and adaptable to other scenarios.

Type System. libErator's type system leverages previous works insights [5, 22], which are summarized in Table 1. Specifically, libErator trivially handles primitive types (*e.g.,* `int`, `char`). For pointers, we try to infer the length of the underlying array through a data-flow analysis. For dynamically sized arrays, the driver allocates a buffer (*i.e.,* through `malloc`), we extend this approach to multidimensional arrays. We verify if some function argument of specific types (*e.g.,* `uint`, `size_t`) control the size of dynamic allocations, and bound their value to avoid out-of-memory errors [42]. In the case of `char*`-like types, we terminate the buffer with `NULL`. Additionally, through data-flow analysis, we infer if chars arrays are used as file path in known system functions (*e.g.,* `fread`), and, in such case, allocate a temporary file to store the fuzz input. Moreover, libErator handles *complete* and *incomplete* structures. *Incomplete* types lack information regarding their size [33], and therefore cannot be allocated. Consumers can only have pointers to *incomplete* types and use the library's APIs to handle their lifetime. *E.g.*, at Line 13 in Listing 1, the second argument of the function `vpx_codec_dec_init` is incomplete and can only be created through the `VPXD_INTERFACE` function. *Complete* types, instead, may need non-trivial API Chains to be properly initialized, as in the case for `vpx_codec_ctx_t`, which needs a correct sequence of API calls (Line 13). Lastly, we handle function pointers by synthesizing empty stub *callback functions* from the API source code, and using their address in the API function calls.

Table 1. libErator's partition of the type system. The "Source" column indicates the source of information necessary for the analysis, while the other columns describe which properties they possess: whether they can be referenced (R), allocated (A), or initialized from fuzzing input (I).

Type	R	A	I	Source
Primitive Types	✓	✓	✓	Header Files
Primitive Arrays	✓	✓	✓	Header Files
Strings	✓	✓	✓	Header Files
File Paths	✓	✓	✓	Library Code
Stub Functions	✓			Header Files
Incomplete Structures	✓			Header Files
Complete Structures	✓	✓	✓	Library Code

Lifetime Properties. Drivers need to handle their internal variable lifetime. Specifically, we support the three variable lifetimes of C: *local*, *dynamic*, and *static*. *Local* variables are allocated in the driver's stack frame and released once the driver terminates its execution. *Dynamic* variables

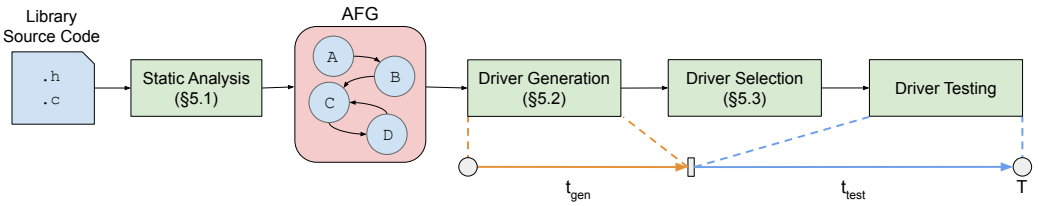


Fig. 2. libErator's architecture. The library source code is analyzed to model the API function arguments and the type system into the API Flow Graph (AFG). Then, the driver generation module synthesizes new drivers and select a diverse set. Finally, the drivers are tested in a fuzzing campaign.

must be allocated and freed coherently. To this end, our analysis mark library functions that *create* variables, *i.e.,* returning allocated structures (*e.g.,* via `malloc`), and functions that *delete* variables, *i.e.,* function arguments passed to `free`. If an API function returns a pointer to a global reference (*e.g.,* BSS), we consider the memory location as *static*. API functions that *create*, *delete*, or return *static* reference hint at inter-API dependency. *E.g.*, an API function that allocates objects should appear early in the driver. Likewise, API functions that `free` objects should appear during cleanup.

Var-Len. libErator supports API arguments dependency between a variable and its length (*Var-len*). Two API arguments, called V and L, are in a *Var-len* relation if V points to buffer and L indicates V's size, *e.g.,* in Listing 1, the function `vpx_codec_decode` requires `data` to be a buffer of bytes (V) of size `frame_size` (L). We ensure that *Var-len* arguments are used coherently. Previous works already investigated *Var-len* properties [5, 23], and we expand on their insights.
Loop-Index: When a `for` or `while` loop traverses a buffer V, the loop index L represents its upper bound (*e.g.,* `while(i<L) V[i]`). To this end, we develop an inter-procedural analysis, while UTopia employs an intra-procedural approach.
Buffer Directly Controlled: Libraries may use the argument L to calculate the last valid address of the buffer V (*e.g.,* `last_V=&V[L]`) and iterate until then. libErator is the first to support these cases.
Third-party Functions: External functions may manipulate buffers, *e.g.,* `memcpy`. If V and L are used as source and size in `memcpy` (*e.g.,* `memcpy(V,data,L)`), then we assume a *Var-len* relation. Our analysis first locates the buffer arguments used in transfer data functions and then checks if any other API function argument is used as a size parameter. Our prototype handles the main `libc` functions known to transfer/alter buffers, *e.g.,* `strcpy`, `memcpy`, and `memset`, and allows for the integration of third-party functions.

5 libErator Design

libErator's end goal is to produce fuzz drivers that (i) can be used in production, and (ii) exhibit diverse library interactions. libErator's design centers around the challenges introduced in §3. Each challenge is addressed by a dedicated technique implemented as a module, see Figure 2. All the modules refer to a central data structure, called *API Flow Graph* (AFG), that encodes API function information. Specifically, the AFG provides hints for creating long API chains thus addressing challenge C1. The AFG is produced by the static analysis module (§5.1) taking only the library source code as input. By leveraging the AFG, a driver generator module (§5.2) produces new drivers through an algorithm that learns to avoid invalid API Chains, *i.e.,* API function sequences that do not adhere to the library semantic. This improves the rate of correct drivers without suffering from coverage wall biases, thus tackling the challenge C2. Then, libErator uses a lightweight clustering algorithm to select drivers that stress different library regions (§5.3), thus enabling longer fuzz

driver testing and satisfying challenge C3. Finally, all selected drivers are fuzzed for an identical period with an initial corpus bootstrapped from the seeds produced during the generation phase.

libErator's configuration only requires the exported library's headers and a build script for the library. The generated fuzz drivers follow the specification in §4.

5.1 Static Analysis

The purpose of the static analysis is to populate the *API Flow Graph* (AFG), the directed graph encoding the API function dependencies and helps predict interesting API Chains. We use a static data-flow analysis to infer non-trivial properties from the library code. When re-using existing techniques, we point to the implementation (§6) and otherwise describe our analysis in details.

API Flow Graph (AFG). The graph is built from the function signatures, exposed in the library header files. The AFG represents an over-approximation of the possible *API Chains* [5, 22, 51], and guides the driver generation (§5.2). Specifically, the graph's nodes represent an API function, and their outgoing edges are possible subsequent API function calls. For each API function A, we define its inputs and outputs, called $I(A)$ and $O(A)$, respectively. Inputs $I(A)$ are the arguments passed to the function, while output $O(A)$ are the return values and the arguments passed by reference. Then, we define B *depends_on* A if the types of $O(A)$ and the types of $I(B)$ have a non-empty intersection, *i.e.,* B *depends_on* $A \iff O(A) \cap I(B) \neq \emptyset$. Based on the *depends_on* relation, we connect node B to node A in the AFG. By traversing the AFG, libErator chooses the next function to append to the API Chain. Then, through static and dynamic analysis, the driver generator module prunes invalid API Chains. In the following, we detail the additional information contained in the AFG.

AFG Bias. libErator uses information obtained from the static analysis to guide (bias) the API Chain creation (§5.2). We observe that libraries commonly use simple functions as preliminary steps for more complex library interactions later on. For instance, `LibHTP`'s drivers require at least three API functions to set up a valid `htp_connp_t` structure necessary for interaction with more complex library functionalities. Unfortunately, the API functions used to build up `htp_connp_t` reach only a shallow coverage. Therefore, state-of-the-art driver generation approaches leveraging code coverage as feedback may not prioritize these functions. Conversely, our static analysis infers signal from the source code as functions setting up more complex library usage manipulate the fields of the desired structure. In particular, we enumerate all the fields and subfields that are set or retrieved, carefully handling recursive structures. Primitive types are assimilated to structures with a single field. The total number of manipulated fields is used to bias the driver generation module when traversing the AFG. By observing how a function manipulates a structure, libErator can foresee more complex API Chains.

Algorithm 1: Driver Creation

```
Input: The AFG: D, Time for generating drivers: t_gen
Output: A list of drivers to be deeply tested
drivers_history ← {};
while time_spent() < t_gen do
    driver ← generate_api_chain(D, drivers_history);
    status ← probe(driver);
    drivers_history ←_add (driver, status);
end while
return extract_positive_drivers(drivers_history)
```

5.2 Driver Generation

The driver generation module relies on the AFG to generate functional drivers (§2). More specifically, the module uses a dynamic generation strategy, in which, increasingly long API Chains are generated and tested. Algorithm 1 describes the overall strategy. Crucially, each new API Chain is probed in a short fuzzing campaign (*i.e., five* minutes). The probing reveals if the driver interacts with the library, *i.e.,* it produces *seeds*. Drivers producing seeds are marked as *positive*, while the others are marked as *negative*. All the API Chains are recorded in a *driver history* that is used in combination

with the AFG for generation. This mechanism overcomes the limitation of previous works that use coverage to promote drivers. Since drivers often face a coverage wall, short fuzzing campaigns cannot faithfully assess drivers' quality. Conversely, malformed drivers hardly show any interaction (seed), giving more reliable feedback. Furthermore, negative chains allow the system to preemptively avoid AFG paths leading to useless drivers. This algorithm "learns" valid API Chains and discards unfruitful ones, thus, addressing challenge C2. Crucially, choosing seeds over coverage also boosts performance, as computing coverage requires resource intensive tools like SanCov [40].

Our algorithm generates an API Chain by traversing the AFG. Throughout the traversal, the algorithm respects the library control- and data-flow constraints encoded in the AFG. During this phase, we use the static analysis results and the *driver history* to bias the AFG traversal towards interesting chains, while avoiding repeating malformed drivers. Then, we produce the code for both the input transfer and cleanup regions of the driver.

API Chain Creation. Algorithm 2 describes the process, which takes the AFG and the *driver history* as inputs. In the first part, libErator identify *source* functions (Line 1–5). We define an API function as *source* if all its arguments can be readily allocated and initialized. *Source* functions represent the AFG entry points. Consequently, we traverse the AFG and try to append new function calls to the chain (Line 7–16). Additionally, we use the *driver history* to discard chains already evaluated as malformed. The algorithm retrieves from the AFG the number of manipulated fields to bias its selection toward instantiable API functions. If no candidate API function is available, it selects a new *source* function (Line 17–22). The algorithm terminates when it creates a new chain that has never been probed, *i.e.,* it is neither *positive* nor *negative* (Line 24). The idea is to start by exploring the *source* functions and then expand while avoiding chains known to be malformed.

The AFG traversal is controlled by an instantiation routine (*i.e.,* `try_to_instantiate`), which is an oracle that answers the question: *Can we invoke the given API function given the current variables (var_alive)?* The instantiation routine decides which API functions can be appended to a given *API Chain*. Formally speaking, the routine has *three* duties: (1) try to reuse already instantiated variables, (2) generate new variables if their type allows it, (3) update `var_alive` to track the variable lifecycle (*e.g.,* for the final cleanup). More importantly, the instantiation routine traces the variable lifetimes and avoids reusing the deallocated ones. Additionally, it coherently associates variables used in *Var-len* arguments and uses additional variables to handle variadic arguments [7].

5.3 Driver Selection

The driver generation module (§5.2) produces a list of *positive* drivers (Line 7). These harnesses are sufficiently stable to be tested, however, can be numerous with up to 100 drivers generated per hour depending on the library. Deep testing each of them is, therefore, infeasible. This is even more problematic when the time budget is constrained, for example during a fuzzer evaluation.

Therefore, the strategy to select the most relevant drivers should satisfy the following requirements. First, it needs to be lightweight. Second, it should select drivers diversifying library usage. While coverage distinguishes drivers that reach different code regions, the short fuzzing period is insufficient to overcome coverage walls, making coverage an unreliable metric. Moreover, calculating fine-grained coverage is resource intensive, stealing resources from the actual testing.

To address this problem, we devise an algorithm around the intuition that API functions are a proxy for library regions. Specifically, we employ a lightweight clustering algorithm based on affinity propagation algorithm [12] and Levenshtein distance [46] to automatically group drivers based on the API functions they use. We treat the API Chains as a list of symbols, where each symbol is an API function, and calculate the Levenshtein distance between each pair of chains. Then, the affinity propagation automatically clusters chains with closer distance. For each cluster, we

Algorithm 2: API chain creation

```
   Input: The AFG: D, and drivers_history
   Output: The API chain and the list of variables
1  source_api ← get_source_api(D);
2  good_api_chains, bad_api_chains ← split_positive_negative_chains(drivers_history);
3  api ← random_bias(source_api, D);
4  api_chain = [api];
5  var_alive ← try_to_instantiate(api, ∅);
6  while true do
7      candidate_next_api_fun = []; /* Search API functions to add to the chain      */
8      foreach n_api ∈ D[api].adjacency_list do
9          if api_chain ∪ [n_api] ∈ bad_api_chains then
10             continue; /* Avoid repeating known failed drivers                     */
11         end if
12         new_var ← try_to_instantiate(n_api, var_alive);
13         if is_valid(new_var) then
14             candidate_next_api_fun ←_add (n_api, new_var);
15         end if
16     end foreach
17     if len(candidate_next_api_fun) == 0 then
18         api ← random_bias(source_api, D); /* If no valid candidates, pick a new source */
19         var_alive ← try_to_instantiate(api, var_alive);
20     else
21         (api, var_alive) ← random_bias(candidate_next_api_fun, D);
22     end if
23     api_chain ← api_chain + [api]; /* Update the chain, if new, stop and probe it */
24     if api_chain ∉ good_api_chains then
25         break;
26     end if
27 end while
28 return (api_chain, var_alive)
```

select the drivers that produced the most seeds, as we deem them more promising for longer testing. Despite affinity propagation having $O(n^2)$ complexity, we process thousands of API Chains in less than a minute, thus fitting our leanness requirement. As a result, our driver selection algorithm finds relevant targets efficiently, maximizing the resources devoted to testing the library and, thus, addressing the challenge C3.

6 Implementation

Prototype implementation. LibErator analysis leverages SVF [38, 45], complemented with 4K LoC of C++ to extract the dependency graph. The rest of the tool is composed of around 5K LoC of Python code. LibErator generates drivers as C programs, subsequently compiled and statically linked with the target library. The drivers' input follow the format from libFuzzer. LibErator, however, can be used with other fuzzing engines, *e.g.,* AFL++.

Static analysis. The static analysis used in §4 and §5.1 is based on the default Use-Def graph provided by SVF [38, 45] using SVF's Flow-Sensitive point-to analysis [19]. However, we observe that the SVF analyses do not correctly resolve indirect calls for global function pointers. Usually, global function pointers are used to set at runtime system-dependent functions (*e.g.,* `malloc`/`free`) through specific environment variables. This limitation leads to an incomplete call graph that hides

Table 2. Targets selected for LIBERATOR evaluation. "#API Func." denotes the number of exposed API functions in the library. The last column reports the duration of LIBERATOR static analysis.

Name	K LoC	#API Func.	Duration
c-ares	55.99	61	1 min 1 s
cJSON	16.57	78	24 s
cpu_features	8.36	7	43 s
libaom	518.38	47	2 h 40 min 39 s
libdwarf	126.83	333	2 h 51 min 26 s
LibHTP	38.59	249	13 min 43 s
libpcap	45.59	89	42 s
libplist	11.25	101	47 s
libsndfile	56.42	40	38 min 43 s
LibTIFF	87.16	196	7 h 32 min 5 s
libucl	17.04	125	2 h 38 min 25 s
libvpx	362.05	38	2 h 19 min 43 s
minijail	18.87	95	20 s
pthreadpool	12.69	30	1 min 27 s
zlib	29.94	88	21 s

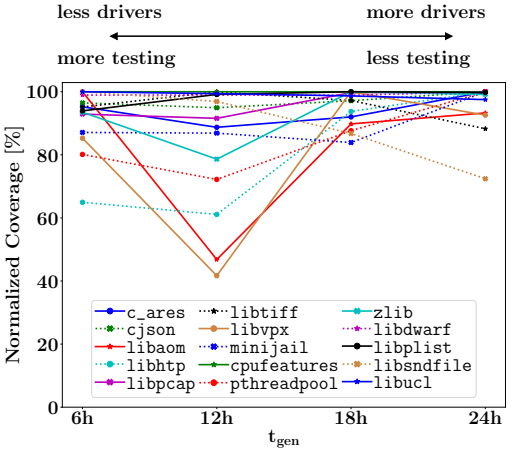


Fig. 3. Normalized coverage in five-runs average achieved by LIBERATOR drivers as a function of t_{gen}. The different behaviors of the libraries as t_{gen} grows, highlight the need for a tool where t_{gen} and t_{test} can be fine-tuned. The raw numbers are in Table 4.

important code patterns. To cope with this issue, we locate (a) all the global structures containing function pointers, and (b) all the code locations where the global function pointers were set. Finally, we use this information to infer missing target sets and update the call graph accordingly. This simple strategy is sufficient to handle the analyzed libraries.

Type Length Value (TLV) implementation. The drivers synthesized by LIBERATOR assume that some variables have a fixed size (*e.g.,* `char s[10]`) while others have a dynamic size (*e.g., Var-len*). On the other hand, the fuzzing engine produces "random" inputs. Therefore, the fuzzer engine and driver must agree on an input structure to correctly mutate and bridge it into variables. To solve this problem, we encode inputs as a TLV structure. More precisely, the driver contains a custom mutator—`LLVMFuzzerCustomMutator` routine in our prototype [16]—that mutates the input according to the TLV encoded structure and maps it to the driver's variables.

7 Evaluation

We evaluate LIBERATOR by answering the following research questions:
RQ1: How does the trade-off between t_{gen} and t_{test} manifest itself in practice (§7.1)?
RQ2: To which extent does LIBERATOR explore libraries? (§7.2)?
RQ3: How does LIBERATOR compare to state-of-the-art library fuzzing tools (§7.3)?
RQ4: Can LIBERATOR find bugs in real-world libraries (§7.4)?
RQ5: How does each component of LIBERATOR contribute to its performance (§7.5)?

Compared Works. Our evaluation compares LIBERATOR against state-of-the-art *consumer-agnostic*—Hopper [5]—and *consumer-dependent* tools—UTOPIA [23], FuzzGen [22], and the Google framework OSS-Fuzz-Gen [20, 21]. We select these works based on the availability of either their artifact or their released drivers. Additionally, we select all manually written drivers from the OSS-Fuzz project [37] and the projects' repositories to provide a comparison with existing drivers.

Benchmarks Selected. We evaluate LIBERATOR on 15 libraries ranging from 8K to 518K LoC, as listed in Table 2. We choose all C targets from Hopper and UTOPIA apart from five libraries which SVF does not support (§8). Additionally, we include the four libraries from OSS-Fuzz-Gen with the most drivers. All the libraries are tested on their most recent commits except when comparing with UTOPIA where we use the versions from their evaluation since the artifact is otherwise incompatible.

Experimental Setup. LIBERATOR evaluation was performed in Docker containers based on Ubuntu 20.04 on an Intel Xeon Gold 5218 @ 2.30 GHz CPU with 64 GB of RAM. Every library and driver is compiled with identical options. Each result is the average of *five* experiment repetitions.

Fuzzing Setup. For all fuzzing campaigns, we instrument libraries and drivers with ASan [36] and SanCov [40]. To measure the coverage, we replay the corpus and count only the branches and lines traced by SanCov in the library while discarding the coverage in the driver itself. Finally, all the testing campaigns are launched with an empty initial corpus.

7.1 RQ1 - t_{gen} vs t_{test} Trade-off Analysis

Automated library fuzzing faces an inherent trade-off between the computation time invested in generating new drivers and the time spent testing them. To demonstrate and quantify this trade-off, we evaluate the overall performance—in terms of code coverage—of fuzzing campaigns with different balances of t_{gen} and t_{test}. In particular, we perform *four* 24-hour campaigns for t_{gen} values of 6-, 12-, 18-, and 24-hours. As the time necessary for selection is minimal, t_{test} is the complement to 24 hours of t_{gen}. If t_{test} is *zero*, we measure the cumulative SanCov coverage produced by the drivers during the driver generation. Otherwise, the coverage is measured during the fuzzing campaign lasting t_{test} hours.

The results of these campaigns are presented in Figure 3. Our main observation is of distinct behaviors among the tested libraries. While `minijail` and `cJSON` reach more coverage while increasing t_{gen}, `libaom` and `LibTIFF` show better results for the smallest t_{gen}, *i.e.,* 6 hours. This correlates with the complexity of inputs accepted by the libraries. For instance, the core of both `LibTIFF` and `libaom` is to parse complex media formats, which the fuzzer is unlikely to synthesize correctly initially. With longer testing time, the fuzzer learns the input format and provide better inputs to the drivers. `cJSON` and `minijail` inputs have simpler structures, and testing benefits more from a diverse set of drivers which allows for broader coverage. Looking at `libpcap` and `zlib`, the coverage plateaus or decreases beyond a certain t_{gen}. Likely explanations are either a coverage wall or the saturation of the driver corpus—*i.e.,* the campaign does not benefit from new drivers anymore. For `libaom` and `libvpx`, we observe a sharp drop in coverage for a t_{gen} of 12 hours. We hypothesize that this setting exemplifies the worst of both worlds: the fuzzer does not have enough time to learn the input structure, and we lack enough driver diversity to trigger more code paths.

In conclusion, Figure 3 shows the absence of a one-size-fits-all for automated library fuzzing. The optimal balance between t_{gen} and t_{test} depends partially on the API complexity and its input structure. This motivates the need for a tool configurable to any balance between t_{gen} and t_{test}. LIBERATOR is designed around controlling this trade-off and can, therefore, be used to find the best resource allocation for fuzzing a specific library.

7.2 RQ2 - How does LIBERATOR Test Libraries?

We investigate how LIBERATOR performs library testing and highlight different aspects of the *valid* drivers synthesized during the generation process (§5.2). To this end, we measure the cumulative coverage and the API functions invoked during a campaign where t_{gen} is set to 24 hours. Specifically, the cumulative coverage is calculated by merging the progressive SanCov profiles [39]. Both metrics

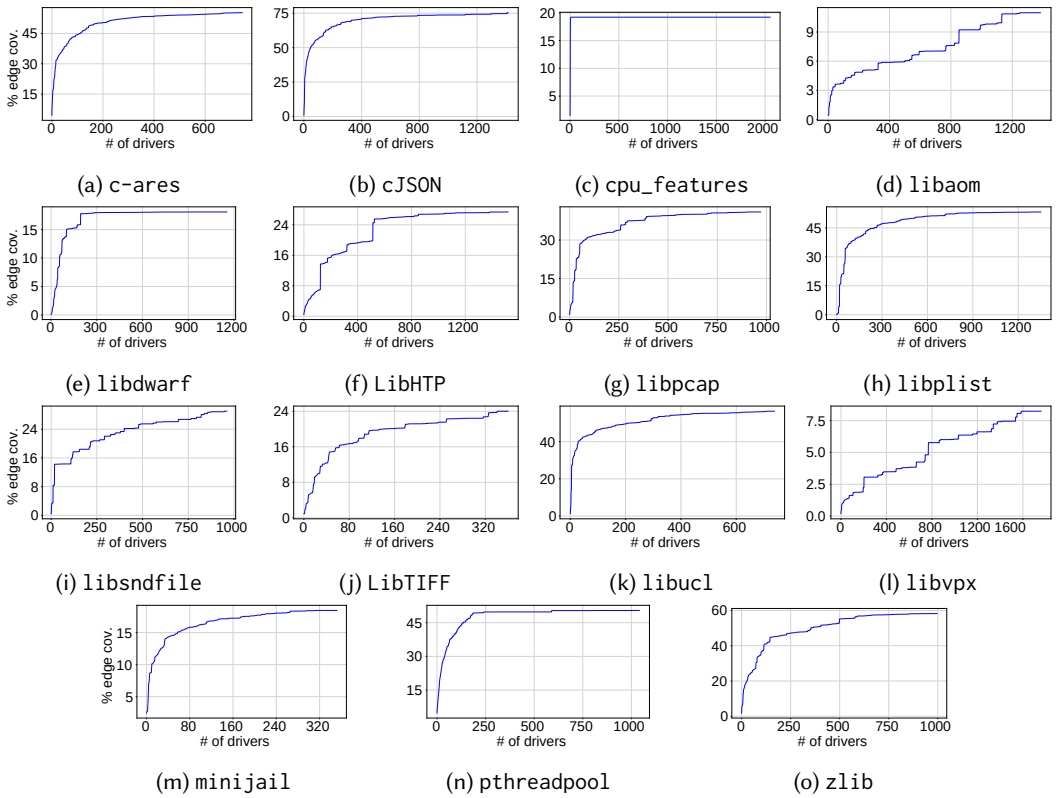


Fig. 4. Cumulative edge coverage reached as drivers are generated over 24 hours. The X-axis is the number of drivers generated during the 24 hours fuzzing session. The Y-axis is the cumulated edge coverage reached.

are expressed in terms of number of drivers generated. If a repetition has fewer drivers, we take its total cumulative coverage for the outstanding driver average.

Figure 4 shows the cumulative coverage after 24 hours. Similarly to standard fuzzing, the coverage tends, for most libraries, to reach a plateau. This signifies that generating additional drivers would, likely, bring only marginal new coverage. This plateau can either highlight the need to switch to more prolonged fuzzing or a coverage wall. Therefore, switching to prolonged fuzzing may not always be beneficial, as will show §7.3. Libraries that expect simple inputs (*e.g.,* integers or strings) benefit more from shorter campaigns with different drivers, as sampling interesting inputs for these types is simpler than for complex structures.

Figure 5 shows the percentage of API functions tested during the driver generation. As expected, the libraries showing a clear plateau in terms of coverage (Figure 4), *e.g.,* `cJSON`, also tend to have exhausted the API functions. Conversely, `minijail`, `LibTIFF`, and `LibHTP` did not reach a clear plateau and libErator might explore new API Chains combinations given more time. Notably, libraries requiring complex inputs and complex API Chains, such as `LibTIFF` and `libaom`, benefit from both a longer driver generation and longer testing.

7.3 RQ3 - Comparison with State-of-the-art

To assess libErator, we compare against publicly released drivers and functional artifacts from state-of-the-art library fuzzing tools. The five works selected have explored library fuzzing from

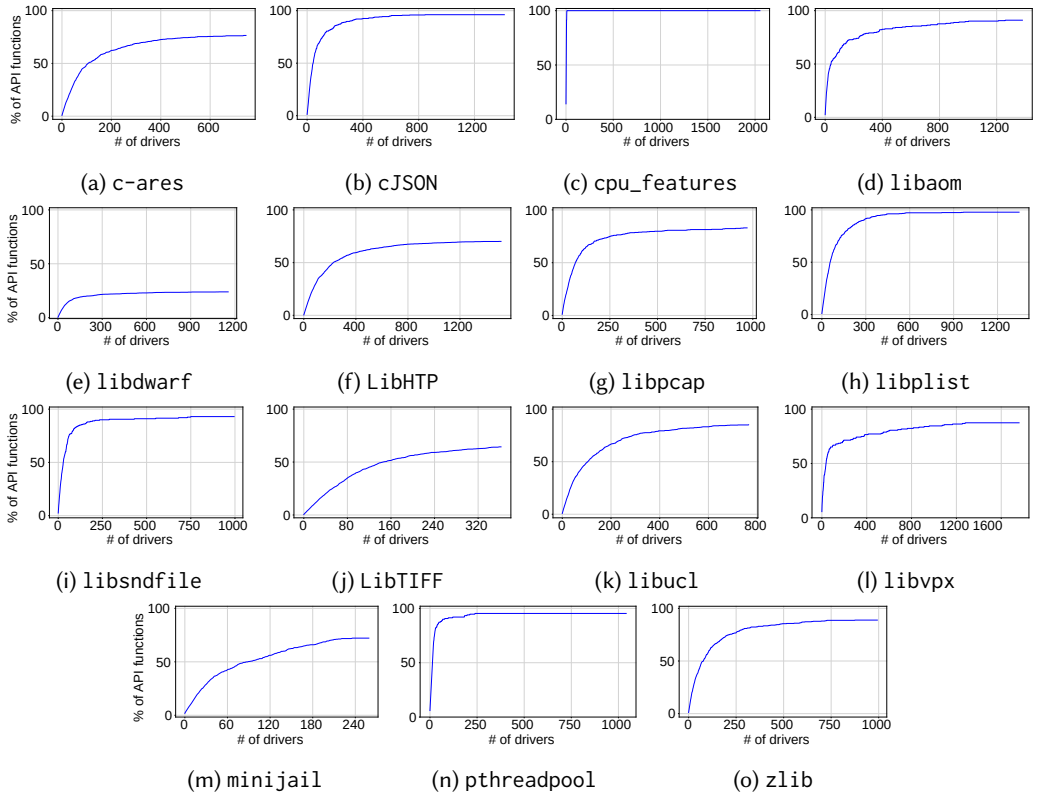


Fig. 5. API function coverage over 24 hours driver generation. The X-axis is the number of drivers selected during the generation. The Y-axis is the percentage of API functions covered as more drivers are created.

different perspectives and with varying assumptions. Table 3 summarizes their characteristics and highlights their differences. Most tools share characteristics, *i.e.,* they operate at source code level, are *consumer-dependent*, and use existing sanitizer (*i.e.,* ASan [36]). Only UTopia and Hopper stand out. UTopia transforms libraries' unit tests into fuzz drivers. However, as unit tests often interact directly with internal library functions, it biases the amount of reachable code, as observed in `LibHTP`. Hopper works with stripped binaries, thus prohibiting the use of ASan, and, instead, employs binary re-writing to implement sanitization. Their approach suffers, however, from false negatives (*e.g.,* it misses out-of-bound memory accesses). This leads Hopper to measure coverage past runtime errors in code unreachable with ASan thereby artificially inflating coverage numbers. To measure this impact, we replay Hopper's queue with ASan and observe that 73.39% and 8.05% of the inputs for `libplist` and `libdwarf` respectively, suffer from runtime errors. From our analysis, we conclude that Hopper and libErator operate on different assumptions (open versus closed source) resulting in coverage measurements not directly comparable.

Table 4 shows the results of libErator across different t_{gen} and t_{test} as well as the results of Hopper and OSS-Fuzz. Table 5 provides a comparison against UTopia, while and Table 6 compares libErator to FuzzGen and OSS-Fuzz-Gen. The following paragraphs describe each comparison in detail. The coverage reported is only the library coverage after a time budget of 24 hours. We select the best libErator configuration for each library and compare its coverage with the other tools.

Table 3. libErator's features compared with state-of-the-art works. UTopia and Hopper are the two main outsiders. UTopia does not adhere to the fuzz driver definition and operates on library functions outside of the public API. Hopper works at the binary level and does not use ASan but an incomplete custom sanitizer.

Tool	Consumer Relation	Standard Sanitizer	Use API Only
UTopia	Dependent	✓	
OSS-Fuzz	Dependent	✓	✓
OSS-Fuzz-Gen	Dependent	✓	✓
FuzzGen	Dependent	✓	✓
Hopper	Agnostic		✓
libErator	Agnostic	✓	✓

Hopper. Despite the difference in sanitization, libErator outperforms Hopper on eight out of 13 libraries. We carefully examine the five libraries where Hopper prevails. In general, Hopper benefits from its faster generation of drivers. This affects the exploration of `LibHTP`, which is the largest library tested in terms of numbers of API functions in our evaluation set (see Figure 2). By generating vastly more API Chains than libErator, Hopper can reach shallow coverage in many more functions resulting in higher coverage overall. This also explains the reached coverage in `c-ares` and `libucl`. For `cJSON`, we observe that Hopper avoids a recurrent false positive use-after-free error that hinders libErator from reaching a higher coverage. For `libplist`, Hopper's sanitizer does not detect a runtime error, thus not stopping library testing. Indeed, we observe that 73.39% of the inputs generated by Hopper for `libplist` would be stopped in libErator by ASan.

OSS-Fuzz. Compared to OSS-Fuzz, libErator covers more code on six of the 12 compatible libraries. For complex APIs like `libaom` and `libvpx`, drivers need complex code structures such as loops and conditions for probing deeper code paths. This limitation is common to all other automatic approaches, *i.e.,* Hopper, OSS-Fuzz-Gen, FuzzGen, and UTopia. OSS-Fuzz's drawback is that the current drivers are exhaustively tested and without extensive manual efforts can hardly reveal new bugs, as we study in §7.4.

UTopia. Table 5 shows the comparison with UTopia. As UTopia converts existing unit tests into harnesses, its drivers interact through library functions absent from the public API, thus giving a substantial advantage in terms of coverage. This is particularly evident for `minijail` and `LibHTP`. For `libaom`, UTopia's drivers are more stable since they include loops and more complex code structures. Nonetheless, libErator overcomes UTopia in half of the targets without requiring any unit tests and respecting the intended library interaction. Moreover, our evaluation of UTopia results only in false positive crashes which are cumbersome to triage.

OSS-Fuzz-Gen. Table 6 reports the comparison with OSS-Fuzz-Gen. Since we lack the resources to run the Large Language Model (LLM), we test the publicly released drivers [21]. Overall, libErator outperfoms OSS-Fuzz-Gen on five out of six libraries while coming close in the remaining one. OSS-Fuzz-Gen is penalized by the small number of API functions included in its drivers. Indeed, the prompts used insist on keeping the driver concise to avoid compilation errors. Conversely, libErator benefits from interaction with diverse API functions thanks to §5.3, thus exercising more diverse parts of the library's code.

FuzzGen. Table 6 presents the results of FuzzGen. Due to compilation errors in the generation pipeline, we fall back to the published drivers overlapping with our target set, namely `libaom` and `libvpx`, note that both drivers required manual patches to compile. Despite our investigation, we

Table 4. Library coverage after 24 hours fuzzing campaign for LIBERATOR, OSS-Fuzz, and Hopper. LIBERATOR results are shown for t_{gen} of 24-, 18-, 12-, and 6-hours, with **bold** for the best configuration. Additionally, we report the coverage delta between the best LIBERATOR configuration and the other tools in the columns marked with Δ. **Bold green** highlights where LIBERATOR prevails. We mark unsupported libraries with a dash.

Target	LIBERATOR [t_{gen} + t_{test}]				Hopper		OSS-Fuzz	
	24h+0h	18h+6h	12h+12h	6h+18h	cov.	Δ	cov.	Δ
c-ares	**55.29%**	50.88%	49.06%	52.65%	60.43%	-5.15%	22.62%	**+32.66%**
cJSON	**75.34%**	73.20%	71.52%	72.68%	87.44%	-12.10%	45.95%	**+29.39%**
cpu_features	**19.22%**	**19.22%**	**19.22%**	**19.22%**	18.06%	**+1.16%**	-	-
libaom	10.98%	10.57%	5.51%	**11.77%**	9.79%	**+1.98%**	44.20%	-32.43%
libdwarf	**18.08%**	17.88%	17.89%	17.89%	11.56%	**+6.51%**	20.04%	-1.96%
LibHTP	**26.74%**	25.07%	16.34%	17.37%	44.14%	-17.41%	31.96%	-5.22%
libpcap	**40.93%**	40.81%	37.48%	38.03%	36.15%	**+4.78%**	43.14%	-2.20%
libplist	54.30%	**54.43%**	53.97%	51.13%	63.07%	-8.63%	35.34%	**+19.10%**
libsndfile	26.50%	31.75%	35.46%	**36.59%**	6.18%	**+30.41%**	11.14%	**+25.45%**
LibTIFF	24.45%	26.93%	**27.70%**	26.36%	-	-	28.33%	-0.62%
libucl	56.34%	57.02%	57.43%	**57.78%**	64.98%	-7.20%	36.88%	**+20.90%**
libvpx	8.24%	**8.90%**	3.71%	7.59%	6.18%	**+2.71%**	48.48%	-39.58%
minijail	**18.45%**	15.48%	16.03%	16.07%	-	-	-	-
pthreadpool	**50.44%**	44.21%	36.42%	40.42%	31.01%	**+19.43%**	-	-
zlib	58.32%	**58.83%**	46.26%	55.01%	38.92%	**+19.91%**	50.14%	**+8.69%**

could not identify a clear cause for the low coverage in libaom. For libvpx, our patch corrected an initialization issue for the vpx_codec_ctx structure, which is essential for library interaction. Resolving this issue allowed FuzzGen to achieve a coverage comparable to LIBERATOR.

Overall, LIBERATOR outperforms Hopper on most libraries and on almost all libraries compared to OSS-Fuzz-Gen and FuzzGen. Compared to UTOPIA, LIBERATOR reports, on average, a comparable coverage despite using only publicly available API functions. Finally, LIBERATOR complements OSS-Fuzz by providing a broader and evolving exploration of the libraries. We therefore conclude that automatically creating high-quality drivers without consumers or test cases is achievable through analysis of the library's source code alone.

7.4 RQ4 - LIBERATOR Bugs Finding Capabilities

We compare the bug-funding capability of LIBERATOR against the previously listed competitors. In particular, Table 7 lists the bugs found by LIBERATOR. We analyze crashes found when testing the target libraries by using the best t_{gen}/t_{test} configuration from §7.3. We denote (with †) additional bugs discovered during LIBERATOR development. Later, we discuss the false positives generated by LIBERATOR and their root causes. Finally, we expand on two case studies about libpcap and cJSON to illustrate the effectiveness of LIBERATOR in finding real-world bugs.

To correctly classify the bugs, we manually inspect their root cause and automatically cluster them in coherent classes via stack similarities [10]. Overall, LIBERATOR identifies 81 unique crashes across all 15 libraries. After triage, we report 24 confirmed bugs resulting in a 25% true positive ratio which is double that of similar state-of-the-art works [23]. Hopper finds only six bugs, while the drivers from OSS-Fuzz, OSS-Fuzz-Gen, FuzzGen, and UTOPIA found *zero* during our evaluation.

True positives: Table 7 list the 24 bugs found by LIBERATOR. Each bug was promptly reported to the maintainers with a fix suggestion. LIBERATOR finds a variety of bugs, including logic errors,

Table 5. libErator comparison against UTopia. The libraries used are from the commits indicated in UTopia. libErator's configurations are from Table 4.

Target	UTopia	libErator *	
	cov.	cov.	t_{gen} + t_{test}
cpu_features	4.45%	**54.97%**	24h + 0h
libaom	**18.66%**	15.61%	6h + 18h
LibHTP	**40.37%**	27.02%	24h + 0h
libvpx	8.41%	**9.31%**	18h + 6h
minijail	**31.59%**	18.73%	24h + 0h
pthreadpool	35.19%	**44.63%**	24h + 0h

Table 6. libErator comparison against FuzzGen, and OSS-Fuzz-Gen. libErator's configurations are from Table 4.

Target	FuzzGen	libErator	
	cov.	cov.	t_{gen} + t_{test}
libaom	0.11%	**11.77%**	6h + 18h
libvpx	**9.35%**	8.90%	18h + 6h
	OSS-Fuzz-Gen	libErator	
cJSON	58.29%	**75.34%**	24h + 0h
libdwarf	**18.66%**	18.08%	24h + 0h
libpcap	1.09%	**40.93%**	24h + 0h
libplist	26.18%	**54.43%**	18h + 6h
libsndfile	0.58%	**36.59%**	6h + 18h
libucl	33.30%	**57.78%**	6h + 18h

Table 7. True positive found by libErator. The table reports the library and the bug type. The † indicates that the bug was found during development (§7.4). The status column indicates if the bug is fixed or only acknowledged by the maintainers. For reference, we report the project issue number. The NULL dereference in libpcap is tracked under CVE-2024-8006.

Library	Bug Type	Status	Issue/PR #
cJSON	Logic error	Fixed	*881*
cJSON	NULL deref.	Fixed	*882*
cJSON	Stack exhaust.	Fixed	*880*
cJSON	Stack exhaust.	Fixed	*880*
cJSON	Stack exhaust. †	Ack.	*827*
cJSON	Logic error †	Fixed	*821*
libpcap	NULL deref.	Fixed	*1345*
LibTIFF	SEGFault	Fixed	*643*
LibTIFF	Int. Overflow	Fixed	*649*
LibTIFF	Int. Overflow	Fixed	*645*
LibTIFF	Arith. Except.	Fixed	*646*
LibTIFF	Arith. Except. †	Fixed	*580*
LibTIFF	Arith. Except. †	Fixed	*628*
libucl	Miss. Check	Fixed	*315*
libucl	Miss. Check	Fixed	*315*
libucl	Miss. Check	Fixed	*315*
libucl	Miss. Check	Fixed	*315*
libucl	Miss. Check	Fixed	*315*
libucl	Miss. Check	Fixed	*315*
libucl	Miss. Check	Fixed	*315*
libucl	OOB Read	Ack.	*316*
pthreadpool	Arith. Except.	Fixed	*46*
pthreadpool	Heap Overflow	Ack.	*47*
zlib	Logic error †	Fixed	*889*

Table 8. Breakdown of false positives generated by libErator. We left out libraries without false positives, namely cpu_features, libplist, and libvpx. *incoherent arg.* indicates incoherency between two or more arguments (*e.g.,* missed *Var-len* relationship). *mem. leak* denotes a memory leak. *use-after-free* points to a free incorrectly tracked. *malloc arg.* indicates unbounded size in a call to malloc. *non init buffer* occurs when uninitialized buffers are used. Finally, *API usage* conveys deviation from the intended usage.

	incoherent arg.	memory-leak	use-after-free	malloc arg.	non-init buffer	API usage	**Total**
c-ares	3	1	1	0	0	0	5
cJSON	1	0	0	0	1	0	2
libaom	1	0	0	0	0	0	1
libdwarf	7	1	1	0	2	1	12
LibHTP	2	0	2	0	0	0	4
libpcap	0	0	0	1	1	0	2
libsndfile	6	1	2	0	0	0	9
LibTIFF	0	0	0	0	2	0	2
libucl	2	1	1	2	1	0	7
minijail	2	1	0	0	0	1	4
pthreadpool	1	0	0	1	0	0	2
zlib	2	2	0	1	0	2	7
Total	27	7	7	5	7	4	57

NULL dereferences, and integer overflows across six libraries. For example, at two locations, `LibTIFF` used to index arrays with signed integers, allowing a negative value to deceive the bounds check. This value would later be interpreted as an unsigned integer, leading to an index out of bound. The logic error in `cJSON` results from incorrect usage of `strcpy` from the C standard library. In `libucl`, libErator triggered seven crashes due to incomplete type checks, and an edge case where a non NULL-terminated string caused an out-of-bound read. Overall, the bugs identified ranged from shallow (*e.g.,* a single API call necessary) to complex (over three chained API calls with inter-procedural dependencies). libErator can adapt to these different scenarios, fuzzing deeper once the shallow API functions are covered.

During our evaluation, Hopper identifies 894 unique crashes but only *six* are true positives. The cause of Hopper's high false positive rate is the imprecision of its sanitizer. Additionally, Hopper's oracle overlooks the object lifecycle leading to even more false negatives. Two true positives were in `pthreadpool` and four in `libucl`, all of which were also identified by libErator.

Our fuzzing of the OSS-Fuzz drivers leads to *zero* crashes, likely due to the exhaustive testing that these drivers already experienced as part of the continuous campaigns lead by Google. This highlights the need for a broader and continuously evolving set of drivers. The bugs found by libErator prove that OSS-Fuzz drivers are not exhaustive and that libErator complements these harnesses leading to the discovery of new bugs.

UTopia, OSS-Fuzz-Gen, and FuzzGen fail to produce any true positive during our experiments. As we were unable to generate drivers for both OSS-Fuzz-Gen and FuzzGen, the publicly available harnesses have likely already been extensively tested. Notably, the OSS-Fuzz-Gen drivers for `cJSON` specifically target an API function—`cJSON_duplicate`—in which libErator found a bug. However, OSS-Fuzz-Gen's drivers are unable to trigger it because they all exercise the same sequence of API functions commonly used in consumers while libErator finds alternative sequences. To further showcase the effectiveness of libErator, we empirically confirm its capability to find the two bugs identified by OSS-Fuzz-Gen in previous versions of `cJSON` and `libplist`.[1] The drivers generated by UTopia, despite their good coverage during our evaluation, produce only false positive highlighting the limitation of basing the generation on existing test suites.

In addition to fixes for these bugs, some libErator drivers were also adapted as test cases for the libraries. For example, we contributed *two* test cases for the `cJSON` library. Following the maintainers' demand, we also contributed libErator drivers to `libpcap` for integration into their fuzzing campaigns. This demonstrates the capacity of libErator to produce valid and interesting drivers, complementary to the existing manual efforts of the maintainers.

libErator exhibits a high true positive ratio, *e.g.,* much higher than what we observed for Hopper and double the one reported by UTopia, and finds real-world bugs in a variety of thoroughly tested libraries, demonstrating its applicability.

False positives: We present libErator's false positives in Table 8. They can be grouped into two broad categories: (a) incoherent arguments (*e.g.,* when libErator misses a *Var-len* dependency), and (b) incorrect handling of memory (*e.g.,* missed *Sink* leading to Use-After-Free or unbounded size in `malloc`). These false positives are caused by the imprecision of our static analysis. For example, libErator currently misses the dependency between 2D arrays and their dimensions (*e.g.,* width and height of an image) leading to unwanted out-of-bound accesses. Additional engineering effort could avoid these false positives.

Only 7% of libErator false positives are caused by drivers using the library incorrectly. For example, in `zlib`, the documentation states that `inflateReset` should not be called after `deflateInit_`. However, libErator is not able to infer this incompatibility. Understanding such dependencies

[1]Two out-of-bound access bugs: issues #800 in `cJSON` and #244 in `libplist`.

Table 9. Library coverage for a t_{gen} of 24 hours across both *full* libErator and libErator without field bias. In the last column, we report the difference between the two, highlighting in **bold green** when libErator prevails.

Target	libErator w/o field bias	*full* libErator	Δ
c-ares	55.48%	55.29%	-0.19 %
cJSON	74.62%	75.34%	**0.71 %**
cpu_features	19.22%	19.22%	**0.00 %**
libaom	8.37 %	10.98%	**2.61 %**
libdwarf	18.15%	18.08%	-0.07 %
LibHTP	10.47%	26.74%	**16.26%**
libpcap	42.25%	40.93%	-1.32 %
libplist	52.60%	54.30%	**1.71%**
libsndfile	21.76%	26.50%	**4.74 %**
LibTIFF	20.65%	24.45%	**3.80 %**
libucl	53.53%	56.34%	**2.81 %**
libvpx	7.83 %	8.24 %	**0.41 %**
minijail	19.53%	18.45%	-1.07 %
pthreadpool	47.96%	50.44%	**2.48 %**
zlib	61.27%	58.32%	-2.95 %

Table 10. The first two columns report the number of drivers generated in 24 hours and how many are selected for fuzzing. The last column shows the duration of this process. Driver selection drastically reduces the number of drivers to test while requiring negligible resources.

Target	Tot. drv	Sel. drv	Avg [s]
c-ares	746	50	0.71
cJSON	1343	102	5.09
cpu_features	2046	7	13.14
libaom	1323	85	3.15
libdwarf	1173	101	3.09
LibHTP	1476	110	6.49
libpcap	931	57	1.72
libplist	2404	25	0.10
libsndfile	970	81	1.76
LibTIFF	612	45	0.81
libucl	910	64	1.06
libvpx	1624	86	7.40
minijail	300	19	5.61
pthreadpool	255	22	0.70
zlib	948	73	3.46

would require more complex static analysis or information external to the source code (*e.g.,* annotations or documentation). We leave this as future work.

Case study: cJSON is an lightweight JSON parser. In addition to straight forward NULL dereference, libErator found logic bugs, for example the function cJSON_SetValuestring could pass overlapping strings to strcpy, which is explicitly disallowed. The maintainers acknowledged and fixed this bug. libErator also find multiple stack exhaustion in cJSON caused by incorrect handling of circular references in JSON objects. Lastly, libErator identifies inconsistencies in how cJSON handles arrays and dictionaries. The maintainers acknowledged the bug and are reflecting on how to address it as a fix would break the API compatibility. libErator's cJSON false positives are caused by two missing *Var-len* relationships and missing the identification of a *Sink* function resulting in a freed variable being passed to an API function triggering a Use-After-Free.

Case study: libpcap is a widely used library for packet capture. It is continuously fuzzed through three manually written fuzz drivers as part of the OSS-Fuzz [37] project. Nonetheless, libErator generated novel drivers that resulted in three crash clusters. Both false positives were caused by allocating buffers smaller than the fixed size mandated in the documentation. The last crash was a NULL dereference in pcap_findalldevs_ex when the directory passed is non-existent. The maintainers acknowledged the bug and awarded CVE-2024-8006. We provided a fix that is now upstream. The OSS-Fuzz drivers were not able to find this bug because they are limited to a narrow subset of the whole exposed API and do not test pcap_findalldevs_ex. Following our finding, the libpcap maintainers requested integration of libErator drivers into their fuzzing campaign.

7.5 RQ5 - Ablation Study

We conduct two ablation studies to understand how the components of libErator contribute to its results. First, we evaluate the impact of the AFG bias used to guide the API Chain creation (§5.1)

and to solve challenge C1. The second study aims at understanding the impact of the clustering developed to avoid fuzzing redundant drivers (§5.3), *i.e.,* challenge C3, on the overall performance of libErator. Regarding challenge C2, an experiment excluding this technique is infeasible since it is core to libErator's driver generation, and removing it would distort the whole architecture.

Table 9 shows the coverage reached after 24 hours of driver generation (24 hours t_{gen}) by two libErator configurations. First, we run libErator without biasing the AFG. In practice, we modify the function `random_bias` (line 18 and 21 in Algorithm 2) to pick fully at random an API function out of the possible candidate. The AFG bias shows crucial improvements in the coverage of `LibHTP`. The complex chains necessary to set up minimal state in `LibHTP` are unlikely to be encountered by libErator without biasing the choice of the subsequent API function during driver generation due to the size of the API of `LibHTP`. Coverage-guided driver generation (*e.g.,* Hopper) shows similar limitations due to the limited coverage reached by their API Chains. For the other libraries, AFG bias does not show a particular effect. From our analysis, this is because most API functions interact with a similar number of fields, therefore, reducing the bias effect.

To assess the impact of our driver selection strategy (§5.3), we measure the duration of the affinity propagation clustering across the drivers generated by libErator in 24 hours with a similar setup as the previous study. Results are presented in Table 10. Despite having to cluster more than 2'000 drivers, the duration is negligible for all the libraries. In the worst case, `cpu_features` takes less than 0.02% of the total experiment time. This shows that the driver selection is not a bottleneck in the pipeline of libErator. By reducing the number of drivers by up to 99%, the driver selection is crucial to avoid redundant drivers and distribute the fuzzing resources efficiently.

Overall, each component of libErator contributes to the increased performance shown in §7.2. The AFG bias is crucial to guide the driver generation for libraries with complex setups (*e.g.,* `LibHTP`) while the driver selection is lightweight.

8 Discussion

Future work. So far, libErator cannot generate drivers containing loops. Supporting loop involves an explosion of complexity as libErator must decide which functions should be called repeatedly and how to handle loop termination. With engineering efforts, however, our static analysis could identify the API functions expected to be used in loops based on argument types.

Despite the design concepts being generic and language-agnostic (§5), our prototype targets C code, but we plan an extension to other languages, such as C++ or Python. For example, libErator for C++ could reuse most of the architecture. New constructs like class hierarchies, templates and generics are not yet handled by libErator resulting in an incomplete AFG for C++ code.

Our current static analysis (§5.1) inherits SVF imprecision when following indirect jumps. Ongoing efforts to improve SVF point-to analysis would enhance the AFG (*e.g.,* avoiding the false positives due to missed *Var-len*). Lastly, SVF cannot analyze some libraries due to state explosion.

Comprehensively understanding the different library characteristics influencing its position along the t_{gen} versus t_{test} ratio remains an open question.

Limitations. libErator adds empty stub functions when an API requires function pointers. Understanding which function behavior is expected is a daunting task that would require other sources of information (*e.g.,* consumers or documentation). Additionally, to further increase the probability of synthesizing valid API Chains, we could leverage Large Language Model (LLM). We consider these techniques orthogonal to ours since we aim to show the limitations of one-size-fits-all solutions and the complexity of striking a balance between driver generation and deeper testing.

9 Related Work

Automated library testing. Randoop [32] and EvoSuite [11] pioneered the automatic generation of tests for Java programs. The latter is based on test case mutations and invariants inference to define specifications that can later be verified. Similar work exist for other languages [29, 30]. Documentation and source code comments have also been used to discover software specifications [2]. Lately, LLMs have improved test suites for functions with little coverage [25]. Contrary to these works, libErator generates fuzzing drivers unleashing fuzzing engine on C libraries.

Driver synthesis. FuzzGen [22], Fudge [1], UTopia [23], OSS-Fuzz-Gen [21], PromptFuzz [47], Winnie [24], AFGen [28], IntelliGen [52], TitanFuzz [8], RUBICK [50], Daisy [53], and NEXZZER [26] leverage external knowledge about the API usage (*consumer-dependent*). Either by trimming and cross-pollinating consumers, transforming unit tests, or by querying LLMs, these tools learns from existing usage and are limited in the diversity of the generated drivers. Moreover, as exemplified in APICraft, they focus on driver generation while not exploring trade-offs between generation and testing, and overlook driver selection strategies. Conversely, libErator separates the duty of generating and testing fuzz drivers, showing that the two problems are orthogonal and a single *one-for-all* solution is not achievable as that strongly depends on the library API.

Hopper [5] and GraphFuzz [18] model library testing as a grammar fuzzing problem and, as libErator, are consumer-agnostic. However, both fail to address the challenges in §3. Conversely, libErator helps strike a balance specific to each library, addressing a new angle of this problem.

10 Conclusion

We laid out the computation trade-off between the time spent generating drivers and the resources invested in fuzzing, and limitation it imposes to the current library fuzzing works. We highlighted the necessity for a driver generation technique capable of dividing resources differently for each library, striking a better balance for each library than a one-size-fits-all approach.

We presented libErator, a novel methodology, configurable along this trade-off, to automatically generate fuzz drivers. libErator employs cutting-edge static analysis techniques to infer the semantics of API functions and, then, generate valid fuzz drivers for a target library. We deploy libErator against 15 libraries and compare our results against both the state-of-the-art driver synthesis generators and manually written drivers.

libErator reaches more coverage than other automated tools while finding bugs in extensively tested libraries where manual written drivers find none. We found 24 bugs that were reported to the maintainers. We upstream derived test cases and fuzz drivers to multiple projects.

11 Data Availability

Our code is on GitHub and Zenodo with DOI:10.5281/zenodo.15201791 [41] under the Apache-2.0 license. Due to length of the fuzzing campaign, the coverage data is too large for Zenodo. We, however, release the raw aggregated results as `CSV`. The instructions on how to run the experiments are in the main GitHub repository.

Acknowledgment

This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2 186974, and the German Research Foundation (DFG) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972.

References

[1] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. 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*.

[2] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 242–253. doi:10.1145/3213846.3213872

[3] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

[4] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[5] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1600–1614.

[6] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596.

[7] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. 2001. FormatGuard: Automatic Protection From printf Format String Vulnerabilities.. In *USENIX Security Symposium*, Vol. 91. Washington, DC.

[8] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[10] Ivannikov Institute for System Programming of the Russian Academy of Sciences. 2023. CASR: Crash Analysis and Severity Report. https://github.com/ispras/casr.

[11] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[12] Yasuhiro Fujiwara, Go Irie, and Tomoe Kitahara. 2011. Fast algorithm for affinity propagation. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

[13] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GreyOne: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[14] Patrice Godefroid. 2020. Fuzzing: Using automated testing to identify security bugs in software. https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/

[15] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft. *Queue* 10, 1 (2012), 20–27.

[16] Google. 2020. Structure-Aware Fuzzing with libFuzzer. https://github.com/google/fuzzing/blob/bb05211c12328cb16327bb0d58c0c67a9a44576f/docs/structure-aware-fuzzing.md.

[17] On2 Technologies / Google. 2023. libvpx. https://chromium.googleource.com/webm/libvpx.

[18] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1070–1081. doi:10.1145/3510003.3510228

[19] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 4 (1999), 848–894.

[20] Google inc. 2023. A Framework for Fuzz Target Generation and Evaluation. https://github.com/google/oss-fuzz-gen.

[21] Google inc. 2023. Fuzz target generation using LLMs. https://google.github.io/oss-fuzz/research/llms/target_generation/.

[22] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2271–2287. https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[23] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. UTopia: Automatic Generation of Fuzz Driver using Unit Tests. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2676–2692. doi:10.1109/SP46215.2023.10179394

[24] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*.

[25] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 919–931. doi:10.1109/ICSE48619.2023.00085

[26] Jiayi Lin, Qingyu Zhang, Junzhe Li, Chenxin Sun, Hao Zhou, Changhua Luo, and Chenxiong Qian. 2025. Automatic Library Fuzzing through API Relation Evolvement. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS 2025)*.

[27] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. 2023. VIDEZZO: Dependency-aware Virtual Device Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 3228–3245.

[28] Y. Liu, Y. Wang, T. Bao, X. Jia, Z. Zhang, and P. Su. 2024. AFGen: Whole-Function Fuzzing for Applications and Libraries. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 11–11. doi:10.1109/SP54263.2024.00011

[29] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 168–172. doi:10.1145/3510454.3516829

[30] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. JSeft: Automated javascript unit test generation. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[31] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. {ParmeSan}: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2289–2306.

[32] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[33] Nikolaos S Papaspyrou. 1998. A formal semantics for the C programming language. *Doctoral Disseration. National Technical University of Athens. Athens (Greece)* 15 (1998), 19.

[34] Greg Roelofs. 2023. libpng. http://www.libpng.org/pub/png/libpng.html.

[35] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[37] Kostya Serebryany1. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[38] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[39] LLVM Team. 2013. LLVM profdata merge. https://llvm.org/docs/CommandGuide/llvm-profdata.html#profdata-merge.

[40] The Clang Team. 2023. SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html.

[41] Flavio Toffalini, Nicolas Badoux, Zurab Tsidnadze, and Mathias Payer. 2025. Artifact for libErator. doi:10.5281/zenodo.15201791

[42] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory errors: The past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses: 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings 15*. Springer, 86–106.

[43] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 866–879.

[44] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.

[45] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*. 327–337.

[46] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence* 29, 6 (2007), 1091–1095.

[47] Lyu Yunlong, Xie Yuxuan Chen Peng, and Chen Hao. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery.

[48] Michal Zalewski. 2013. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[49] Hamzeh Zawawy and Jon Bottarini. 2023. Android goes all-in on fuzzing. https://security.googleblog.com/2023/08/android-goes-all-in-on-fuzzing.html

[50] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie, Xiapu Luo, Xinghua Li, Yang Liu, et al. 2023. Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2867–2884. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-cen

[51] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source {SDK} Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. 2811–2828.

[52] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. IntelliGen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 318–327.

[53] Mingrui Zhang, Chijin Zhou, Jianzhong Liu, Mingzhe Wang, Jie Liang, Juan Zhu, and Yu Jiang. 2023. 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)*. 87–98. doi:10.1109/ICSE-SEIP58684.2023.00013

[54] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FishFuzz: catch deeper bugs by throwing larger nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 1343–1360.

Received 2025-02-23; accepted 2025-04-01