WildSync: Automated Fuzzing Harness Synthesis via Wild API Usage Recovery

WEI-CHENG WU, Dartmouth College, USA
STEFAN NAGY, University of Utah, USA
CHRISTOPHE HAUSER, Dartmouth College, USA

Fuzzing stands as one of the most practical techniques for testing software efficiently. When applying fuzzing to software library APIs, high-quality fuzzing harnesses are essential, enabling fuzzers to execute the APIs with precise sequences and function parameters. Although software developers commonly rely on manual efforts to create fuzzing harnesses, there has been a growing interest in automating this process. Existing works are often constrained in scalability and effectiveness due to their reliance on compiler-based analysis or runtime execution traces, which require manual setup and configuration. Our investigation of multiple actively fuzzed libraries reveals that a large number of exported API functions externally used by various open-source projects remain untested by existing harnesses or unit-test files. The lack of testing for these API functions increase the risk of vulnerabilities going undetected, potentially leading to security issues.

In order to address the lack of coverage affecting existing fuzzing methods, we propose a novel approach to automatically generate fuzzing harnesses by extracting usage patterns of untested functions from real-world scenarios, using techniques based on lightweight Abstract Syntax Tree parsing to extract API usage from external source code. Then, we integrate the usage patterns into existing harnesses to construct new ones covering these untested functions. We have implemented a prototype of this concept named WildSync, enabling the automatic synthesis of fuzzing harnesses for C/C++ libraries on OSS-Fuzz. In our experiments, WildSync successfully produced 469 new harnesses for 24 actively fuzzed libraries on OSS-Fuzz, and also 3 widely used libraries that can be later integrated into OSS-Fuzz. This results in a significant increase in test coverage spanning over 1.3k functions and 16k lines of code, while also identifying 7 previously undetected bugs.

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

Additional Key Words and Phrases: Fuzzing, Fuzzing Driver, Fuzzing Harness, Program Synthesis, Software Analysis, Vulnerability Detection

ACM Reference Format:
Wei-Cheng Wu, Stefan Nagy, and Christophe Hauser. 2025. WildSync: Automated Fuzzing Harness Synthesis via Wild API Usage Recovery. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA043 (July 2025), 22 pages. https://doi.org/10.1145/3728918

1 Introduction

Fuzzing is a widely adopted technique for uncovering vulnerabilities in software by feeding the target program with a large number of automatically generated inputs. Because of its simplicity, ease of deployment, and effectiveness in revealing undesired program behaviors, it has gained

Authors' Contact Information: Wei-Cheng Wu, Dartmouth College, Hanover, USA, wei-cheng.wu.gr@dartmouth.edu; Stefan Nagy, University of Utah, Salt Lake City, USA, stefan.nagy@utah.edu; Christophe Hauser, Dartmouth College, Hanover, USA, christophe.hauser@dartmouth.edu.

ACM 2994-970X/2025/7-ARTISSTA043
https://doi.org/10.1145/3728918

attention in both industry and academia ever since its first introduction in the 1990s [18]. Major companies like Google [19], Cisco [1], Microsoft [2], and numerous open-source projects[22] have incorporated fuzzing into their security development practices [17].

In recent years, attention has been paid to the fuzzing of open-source software libraries. As being the backbone of the modern software ecosystem, it is important that the open-source foundation be secure and reliable. In response, Google launched the OSS-Fuzz project [22] in 2016 to provide continuous fuzzing for open-source software libraries. Developers are encouraged to construct harnesses and integrate their libraries with OSS-Fuzz to expand test coverage. To-date, OSS-Fuzz supports multiple programming languages and has identified over 36,000 bugs in more than 1,000 open-source projects.

To conduct fuzzing on software libraries, fuzzing harnesses are essential as they provide entry points for the fuzzer to interact with the library under test. These harnesses essentially consist of programs containing the desired sequences of library API calls for testing purposes. Creating an effective fuzzing harness is not trivial, as it requires the developer to understand a library's API usage and construct harnesses with appropriate function parameters passing to the API call sequences.

Despite the extensive effort dedicated to fuzzing open-source libraries at scale, a significant portion of the code remains uncovered, as highlighted in the OSS-Fuzz report [3]. In our preliminary study, we examined the usage of 50 C/C++ open-source projects tested on OSS-Fuzz in comparison to their external usage. Surprisingly, we discovered over 1,000 library APIs utilized by other projects that are not covered by either OSS-Fuzz or the library's internal test cases. As being publicly accessible APIs, it is reasonable to construct clear and effective harnesses for these untested APIs, not solely relying on the chances of these functions being deeply embedded in the existing harnesses to be accessed by conventional fuzzers. Although multiple tools have been proposed to generate fuzzing harnesses based on the consumer code of the library, most of them rely on compiler-based analysis, such as runtime instrumentation or LLVM-based analysis, on the external projects to extract the API usage [6, 12, 28]. This limits the ability of existing methods to scale up without significant manual intervention.

Unlike prior works that attempt to capture complete API call sequences from external projects, we propose a new approach to automatically synthesize fuzzing harnesses for untested functions. Our approach leverages the usage of these untested functions in real-world consumer code, along with the workflow embedded in existing harnesses. We have observed that many libraries build their core functionalities based on constructing library-specific non-primitive type data structures. Within a library, functions accepting parameters of the same non-primitive data types often exhibit shared data semantics. By utilizing existing harnesses that facilitate the conversion of fuzz inputs into these specialized data structures, we can effectively test previously uncovered functions that also accept the same data structures. Since these core data structures have already been instantiated within the existing harnesses, our analysis can be confined to a small subset of the external codebase to infer the correct API usage patterns. This enables us to identify the necessary preconditions and constraints required to invoke the newly targeted APIs in a manner consistent with their intended usage, without the need for extensive analysis of the whole consumer codebase.

With this approach in mind, we have developed WildSync to automatically synthesize fuzzing harnesses for untested library APIs with their usage in the wild. WildSync begins by searching for external usage of the target library based on the library dependency information acquired from a Linux distribution's package manager. It then extracts the usage of a target library from the source code of the dependent libraries. By relying on data flow analysis based on Abstract Syntax Tree (AST) parsing, WildSync extracts the API usage from the external source code. This lightweight static code analysis approach allows WildSync to scale up easily without requiring compiled

information from the consumer code. With WildSync, we demonstrate that today's ubiquitous software reuse is an opportunity for automating fuzzing harness generation, while also provide a way to examine the correctness of library usage throughout the nowadays complex software supply-chain.

To showcase the effectiveness and practicality of our approach, we applied WildSync to 24 existing C/C++ libraries continuously fuzzed on OSS-Fuzz. WildSync automatically searched for untested API usage in the wild and synthesized new fuzzing harnesses for these libraries. We also selected 3 widely used open-source libraries currently not on OSS-Fuzz to demonstrate the workflow of using WildSync to generate harnesses from scratch. With the aid of WildSync, 469 new harnesses were created, which achieves up to 400% improvement in test coverage for these libraries. Despite the fact that most libraries have been extensively fuzzed on OSS-Fuzz, we are able to identify 7 new bugs with the new harnesses generated by WildSync after a manual review of the results.

In summary, this paper makes the following contributions:

- We propose a new approach to automatically generate fuzzing harnesses by leveraging the usage of untested functions from real-world scenarios. Our approach is purely a language-level technique, without the need for non-standard build artifacts or specialized but often-unavailable testing frameworks.
- We have developed an automatic harness synthesis tool, WildSync, designed to generate fuzzing harnesses for C/C++ libraries. In our evaluation, WildSync successfully synthesized in total of 469 new harnesses for 27 open-source libraries. This results in an increase of nearly 20k lines of code coverage and the discovery of 7 new bugs.
- We make WildSync publicly available at: https://github.com/spencerwuwu/WildSync.

2 Background Knowledge & Challenges

2.1 Background Knowledge

Fuzzing refers to testing a program by providing it with random or semi-random data as input. A fuzzer begins by executing the program with an initial set of seed inputs. While executing the program, the fuzzer monitors the code coverage and generates new inputs likely to reach unexplored code areas. These new inputs are then executed, providing additional information for the fuzzer to generate further inputs.

To conduct fuzzing on software libraries, fuzzing harnesses play a crucial role in enabling fuzz testing of software libraries by serving as designated entry points through which the fuzzer interacts with the library under test. A conventional fuzzing harness typically consists of the following structure:

(1) **Init:** Consume a data buffer (fuzz input) and initialize the corresponding data structure for the library APIs based on the input data.
(2) **Process:** Validate the correctness of the given data format and execute multiple operations with the library APIs that developers wish to test.
(3) **Destroy:** Destroy any data structures created and free up memory space.

2.2 Challenges for Synthesizing Fuzzing Harnesses

While the concept of fuzzing may seem straightforward, creating effective fuzzing harnesses for libraries at scale is far from trivial. Developers encounter several challenges when creating effective fuzzing entry programs, whether done manually or automatically:

- **API call sequences:** The harness should invoke API sequences in a manner that accurately reflects how the target library should be used, ensuring efficient testing of library usage.

- **Function parameters:** Constructing harnesses requires careful consideration of the appropriate function parameters to pass to the API call sequences.

Multiple research works [6, 12, 26, 28, 29] have been presented to try to capture the usage of these API sequences from various sources of consumer code and convert them into harnesses. However, existing implementations often rely on compiler-based analysis of external projects, which can be limited to specific environments or difficult to scale up. Even when compiled information is available, the extracted API sequences may be overly lengthy, or the API usage may be scattered across the codebase, making it challenging to track relations. Several tools even require developers to fine-tune the harnesses manually [6, 12], or to provide additional information to guide the generation process [13, 14], which creates additional overhead and may not be feasible for large-scale automation.

Another direction of the proposed works involves analyzing the internal code of the library [8, 11, 15]. Harnesses generated by these tools could be biased on the internal developers' understanding of the library, or being diverged from how the library is actually used in practice.

3 WildSync's Approach

```
// int ov_time_seek(OggVorbis_File *vf, double s);

bool VorbisDecoder::seek(double s)
{
   int result = 0;

// Avoid ov_time_seek (which calls ov_pcm_seek) when seeking to 0, to avoid
// a bug in libvorbis <= 1.3.4 when seeking to PCM 0 in multiplexed streams.
   if (s <= 0.000001)
      // ....
   else
      result = ov_time_seek(&handle, s);
    // ...
```

Listing 1. External usage of `ov_time_seek`

The design of WildSync originated from the observation that many libraries construct their core functionalities using custom non-primitive type data structures. Within a library, functions accepting parameters of the same non-primitive data types often imply commonalities in their usage patterns. For instance, the `gdImagePtr` data structure in the `libgd` library is utilized across multiple API functions to manipulate images, while `struct archive` and `struct archive_entry` in `libarchive` handle all archive file operations.

As the core purpose of fuzzing harnesses is to convert fuzzing inputs into these data structures, and then exercise the library APIs with them. For the libraries that exhibit such patterns, we can leverage the existing harnesses that initialize the data structures and test more APIs that accept the same data structures.

To achieve this, we group API functions based on the data structures they accept and further categorize them into 3 types of functions as discussed in Section 2.1: *init*, *process*, and *clean-up*. As *init* and *clean-up* functions are responsible for setting up the fuzzing process correctly, we consider them non-trivial to automatically generate harnesses for, which requires careful engineering. Conversely, the *process* type of functions that accept the same function parameters are suitable candidates for substitution or combination and can be fuzzed in a meaningful manner. We propose to synthesize new harnesses for the *process* functions by mimicking the workflow of existing ones, thereby extending the previous efforts conducted by the developers to test the library usage comprehensively.

As these untested APIs serve different functionalities, they naturally require additional parameters not present in the existing harnesses. Moreover, there may be additional constraints that must be

satisfied to call these functions correctly. We propose to extract the external usage of these untested APIs, identify the additional parameters suitable for fuzzing, while adhering to any constraints specified in the external source code. We illustrate this process with Listing 1, which showcases an external code snippet utilizing the `ov_time_seek` function from the `vorbis` library. Assuming that the existing harness of `vorbis` covers the initialization of the `OggVorbis_File` data type, our objective is to extend it to fuzz the `ov_time_seek` function. In the code snippet, we observe that the function `ov_time_seek` is called with an additional double parameter `s`. While we cannot infer the exact value of this parameter from the code snippet alone, we can populate it with fuzzing input to robustly test the function. At the same time, we should also include the condition check applied to the parameter `s` in the external code, so that the API can be called correctly without triggering known limitations or redundant crashes.

With this model in mind, we move away from the traditional approach of capturing and creating complete API call sequences. Instead, we focus on analyzing the surrounding code snippets to extract the necessary semantics for correct API usage. To achieve this, we developed a lightweight analysis tool that extracts the Abstract Syntax Tree (AST) of the relevant code snippets. As long as the analyzed code is syntactically correct, we can extract the required information to generate harnesses for the target API functions. This approach scales efficiently since it eliminates the need to compile the entire codebase for usage extraction — a task that would require significant effort. Instead, we can collect usage patterns at scale, enabling broader and more efficient analysis.

4 Design of WildSync

With the presented idea in mind, we have designed WildSync to automatically synthesize fuzzing harnesses for untested functions. The workflow of WildSync is depicted in Figure 1.

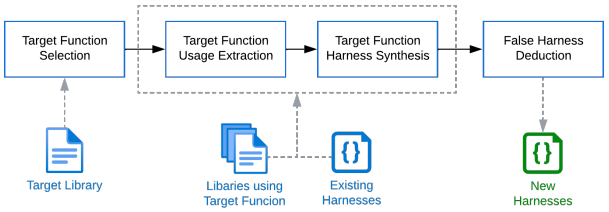


Fig. 1. Workflow of WildSync.

First, WildSync employs several filters to identify target functions for which harnesses need to be synthesized. Subsequently, WildSync extracts the usage of the target functions from the source code of the dependent libraries using Abstract Syntax Tree (AST) parsing and variable data flow analysis. Based on the extracted code, WildSync determines the new variables that need to be fuzzed. After that, WildSync synthesizes new fuzzing harnesses for the untested functions by integrating the extracted usage into new harnesses based on existing ones. A try-compile and try-fuzz procedure is conducted to eliminate invalid harnesses.

Regarding the source of external usage of the target function, WildSync leverages a Linux distribution's package manager and retrieves all other open-source projects[1] that are using the library under test. This source of information is chosen for inferring external usage of libraries due to its reliability and public availability.

[1]Included as packages available for this distribution.

4.1 Target Function Selection

The first step of WildSync is to identify the target functions for which harnesses need to be synthesized from all the untested functions. As described in Section 3, we categorize the library APIs into 3 types: *init*, *process*, and *clean-up* functions. This is done naively based on their names; for example, functions with "create" in their name are categorized as init functions, while those with "destroy" are considered clean-up functions. Currently, we focus solely on synthesizing harnesses for the *process* functions and exclude the other two types of functions. We consider the other two types out of the scope of the current design of WildSync. Error handling and some I/O related functions are also excluded from the synthesis process as they are not the main target of fuzzing. The list of keywords we use to categorize functions can be found in Table 1.

Additionally, WildSync applies a filter to remove uninteresting functions to be fuzzed based on the function's structural complexity and the number of additional function calls it makes. For the current implementation, we keep only functions that have at least one function call, branch, loop, or have more than 5 lines of code in the function body.

Table 1. Keywords to match in function name to group APIs into different categories.

Category	Keywords
Init	create, init, new, alloc, open
Clean-up	destroy, free, close, delete
Excluded	errno, error, print, dump, geterr, cancel, write, pathname, timeout, file, handler, strerror, perror, exporter, display

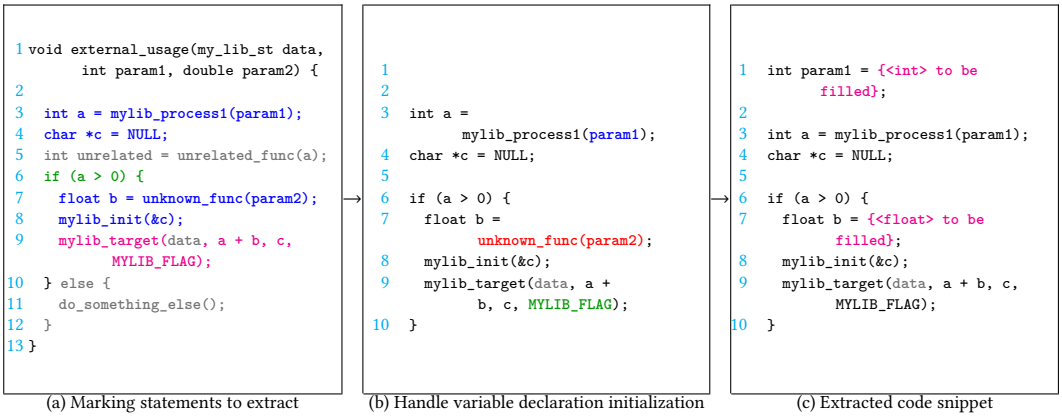
Once the target functions are identified, WildSync matches them with existing harnesses. It extracts all function signatures from the target library's source code and compares the data types of their parameters. Each target function is then paired with the existing harness that initializes the highest number of the same library-specific non-primitive data types used as function parameters. For primitive data types, WildSync in general does not attempt to match them with existing ones and seeks to initialize them with external usage. However, if the function signatures being matched contain not only identical data types but also identical names for these primitive data type parameters, WildSync will instead directly use these values from the existing harness instead of trying to initialize them. Functions without a matching harness are excluded from the synthesis process.

4.2 Target Function Usage Extraction

The purpose of extracting the usage of the target function in real-world scenarios is to capture the correct procedure to set up function parameters that are not initialized in the existing harnesses. Since single external code snippets may contain numerous other libraries or variables that are not locally defined, the current design of WildSync extracts the minimal size of code while retaining the essential initialization and constraints before passing these function parameters to the target function.

Step 1: Marking Data Dependent Statements. To accomplish target function usage extraction, WildSync first constructs a Control Flow Graph (CFG) and a Data Flow Graph (DFG) based on the Abstract Syntax Tree (AST) of the external code snippet calling the target function. Each node in the CFG and DFG represents a statement in the code snippet. For each function parameter not initialized in the existing harnesses, WildSync utilizes a backtracing algorithm on the DFG to obtain the dependency def-use chain from the target function to the first use of this function parameter. The backtracing algorithm is outlined in Algorithm 1. It begins from the target function

call in the DFG, and traces back each parent node. When a parent node has data dependency with the current node on the function parameter currently tracking, WildSync marks the parent node as a required data-dependent statement to be extracted later. The procedure halts upon reaching the first use of this function parameter within the external code snippet, whether it's the declaration of or undefined locally (such as being a global variable).



(a) Marking statements to extract (b) Handle variable declaration initialization (c) Extracted code snippet

Fig. 2. Example of extracting external usage of `mylib_target`. Assuming variable `data` will be provided by the existing harness synthesizing with. (a) marks the depended statements (data-flow dependency marked in blue and control-flow in green). (b) shows the variables that need to be declared and initialized (blue is the variable required declaration; red is an unknown function as an assignee to be removed; green is an identified macro to be kept in place). (c) shows the extracted code snippet.

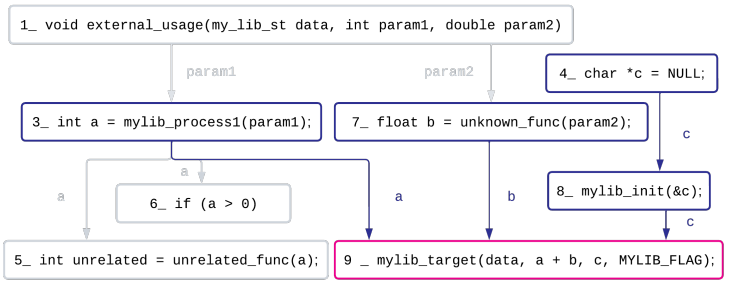


Fig. 3. Data Flow Graph (DFG) of the external code snippet in Figure 2 (a).

This process is illustrated with an example in Figure 2 (a), alongside with the corresponding DFG in Figure 3. In the example, the target function is `mylib_target`, and `data` is a function parameter already initialized in the existing harnesses. The objective here is to extract how this external code initializes other variables passing to `mylib_target`. After applying the backtracing algorithm, WildSync marks the statements at Line 3, 4, 7, and 8, which we highlight in blue. It is noteworthy that the current implementation of WildSync aims to minimize the size of the extracted code snippet. The tracing algorithm only tracks code statements directly altering the value of the function parameters, disregarding any potential side effects for all other code ignored. For instance, Line 5 will be excluded during backtracing from `mylib_target` as it does not alter any value passing to `mylib_target`, despite its association with the `a` variable.

Step 2: Completing Control Flow Structure. After analyzing the data dependency, the next step is to complete the extracted code snippet by addressing CFG dependencies. As the DFG backtracing algorithm only captures the collection of data-dependent statements, WildSync needs to reconstruct the control flow structure among these lines. The algorithm for this process is shown in Algorithm 2. This step aims to capture the control flow structure (e.g. if-else conditions, loops, etc) among the marked data-dependent statements. Similar to the DFG backtracing algorithm, WildSync starts from the target function call in the CFG and traces backward to each parent node. If the edge between the current node and its parent node is a conditional edge, plus there exists a marked data-dependent statement with a line number smaller than the current node, WildSync marks this parent node as a required control statement. The result of applying this algorithm on the example code in Figure 2 (a) is marked in green. Specifically, the `if` condition in Line 6, situated between marked Line 3 and Line 8, is determined to be an essential control statement and thus included in the extracted code snippet.

Algorithm 1 Backtracing from target API in DFG to mark data dependent statements

```
new_nodes ←[target function statement in DFG]
var ← name of the variable backtracing
DFG_marked_nodes ← [ ]
while new_nodes do
    cur_node ← new_nodes.pop()
    for each parent_node of cur_node do
        if has_seen(parent_node) then
            continue
        end if
        if has_data_dependency(parent_node, cur_node,
var) then
            DFG_marked_nodes.add(parent_node)
            new_nodes.add(parent_node)
        end if
    end for
end while
return DFG_marked_nodes
```

Algorithm 2 Backtracing CFG from data dependent statements to mark control flow dependency

```
new_nodes ←[target function statement in CFG]
contational_edges ←[conditional edges e.g. "if_pos", "if_neg"]
CFG_marked_nodes ← [ ]
while new_nodes do
    cur_node ← new_nodess.pop()
    for each parent_node of cur_node do
        if has_seen(parent_node) or
            parent_node.lineno ≥ cur_node.lineno then
            continue                    ▷ Check lineno to avoid loop
        end if
        edge_type ← type of edge from parent_node to cur_node
        if edge_type is conditional_edges and
            ∃ DFG_marked_node.lineno < parent_node.lineno
         then
            CFG_marked_nodes.add(parent_node)
        end if
        new_nodes.add(parent_node)
    end for
end while
return CFG_marked_nodes
```

Step 3: Complete Variable Declaration and initialization. With the extracted code, WildSync proceeds to complete the extracted code snippet by adding missing variable declarations and marking locations required for value assignments. These marked locations will later be assigned with values during the synthesis of the new harnesses. To achieve this, WildSync constructs a new AST and executes the following steps: (1) Removal of unknown functions and (2) Declaration and initialization of unseen variables.

First, WildSync imports the symbols of functions exported by the target library as well as common standard libraries in order to preserve the code sections referencing these functions in the extracted code snippet. Other functions that are not declared within these scopes are directly removed for being unrelated to the current approach. If a removed function acts as an operand to another value assignment, this location is marked for later assignment. Next, WildSync declares and initializes all the variables that are not defined in the extracted code snippet. For declaration, if the variable appears in the external code snippet but is not defined locally, WildSync reinstates the declaration with the same type. If the variable is not defined in the external code snippet, it may be a global variable or a value accessed by pointers and structures, In this case, WildSync simply declares them as random integers or pointers to memory with arbitrary values. After declaring all missing variables, WildSync marks the locations where value initialization is required along with the types to be assigned. For macros, WildSync applies simple heuristics to distinguish them and keeps them as they are in the extracted code snippet.

In Figure 2 (b) and (c), we illustrate the process of completing variable initialization.

4.3 Fuzzing Harness Synthesis

After obtaining the extracted code snippet, we attempt to synthesize new harnesses by integrating the extracted usage with existing ones. First, based on the previous matching candidates described in Section 4.1, WildSync places the extracted code snippet into the existing harnesses at the location where the matched core data structures are initialized. The current strategy is to insert the extracted code snippet into the existing harnesses instead of replacing anything to minimize potential side effects. The variable names of these core data structures within the extracted code are replaced with the ones in the original harness. Variable name collisions are resolved by renaming the variables in the extracted code snippet.

Next, WildSync attempts to provide values for the variables that need initialization in the extracted code snippet. This is achieved by splitting the data bytes provided by the fuzzer into different chunks, and assigning them to the marked locations according to the requested types. As the current implementation of WildSync targets libFuzzer-style harnesses, this is accomplished by rewriting the harness entry point `LLVMFuzzerTestOneInput`. Figure 4 demonstrates the synthesized harness for the example external code snippet from Figure 2. WildSync synthesizes the new harness by inserting the extracted code snippet into the existing harness, and then splitting the original input data buffer into different chunks and assigning them to the marked locations.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *_DATA, size_t _LEN) {
  // Splitting the input data, need an extra <int> and <float>
  if (_LEN < sizeof(int) + sizeof(float) + 1) return 0;
  const uint8_t *_DATA_POINTER = _DATA;

  int __fuzz_0;
  memcpy(&__fuzz_0, _DATA_POINTER, sizeof(int));
  _DATA_POINTER += sizeof(int);

  float __fuzz_1;
  memcpy(&__fuzz_1, _DATA_POINTER, sizeof(float));
  _DATA_POINTER += sizeof(float);

  // End splitting data by redirecting the original function parameters
  const uint8_t *data = _DATA_POINTER;
  size_t len = _LEN - (sizeof(int) + sizeof(float));
  if (len < 1) return 0;
  my_lib_st a = mylib_initdata(data, len);
  if (!a) return 0;

  int param1 = __fuzz_0;
  int _a = mylib_process1(param1); // Fixing name collision
  char *c = NULL;
  if (_a > 0) {
    float b = __fuzz_1;
    mylib_init(&c);
    mylib_target(a, _a + b, c, MYLIB_FLAG); // Subsitute matched variable
  }
  mylib_do_something(a);

  mylib_destroy(a);
  return 0;
}
```

Fig. 4. Example of new harness synthesis from Figure 2. The gray areas are the new synthesized code and white is the original harness. WildSync first splits the original input data into two parts, an integer and a float (marked in red), in order to fill in the values marked in the extracted code snippet. Then insert the rewritten extracted API call.

For simplicity and efficiency when exploring target functions, WildSync synthesizes one harness for each target function instead of combining multiple target functions into one harness.

4.4 Invalid Harness Deduction

In the process of synthesizing new harnesses, WildSync may substitute unknown variable types with integer or arbitrary chunks of data, potentially leading to invalid harnesses that do not properly initialize data structures passed to the target function. Therefore, after all harnesses have been synthesized, WildSync performs a try-compile and try-fuzz step to eliminate these invalid ones.

During the try-compile stage, most of the invalid harnesses are removed as they failed to compile due to inconsistent type casting or other syntax errors. Try-fuzz further provides an opportunity for developers to manually tweak the harnesses, with immediate crash feedback after running the fuzzer. This allows developers to refine the harnesses for APIs that have implicit constraints not captured by the current design of WildSync.

5 Evaluation

In this section, we'll start by providing an overview of the implementation details of WildSync. Next, we'll delve into evaluating its effectiveness in both test coverage and bug discovery.

5.1 Implementation of WildSync

We developed WildSync primarily in Python, comprising approximately 5,000 lines of code in total. At the moment, WildSync focuses on generating harnesses for C-style library APIs, and utilities existing harnesses from libraries hosted on OSS-Fuzz. We retained all the compiling options and initial fuzzing seed selections, making minor modifications to the scripts for each library under test to incorporate the new harnesses.

5.1.1 Target Function Selection. The initial step of WildSync involves identifying the exposed APIs and selects those for which harnesses will be synthesized for. To identify the exposed APIs, all exported function symbols of the library compiled binary are considered as potentially exposed API. Next, WildSync determines the functions it can synthesize harnesses for by comparing the function parameters with those covered in the existing harnesses. As outlined in Section 4.1, *init*, *clean-up*, and error-handling related functions are excluded from selection based on their names.

5.1.2 Target Function Usage Extraction & Harness Synthesis. WildSync retrieves the usage of the target functions from external code sources. This involves fetching the source code of open-source projects that depend on the library under test, leveraging the package management system of `ArchLinux` to identify the dependencies. The core engine of analyzing Control-Flow Graph (CFG) and Data-Flow Graph (DFG) with a lightweight Abstract Syntax Tree (AST) parser is built on top of COMEX [9], a Python wrapper for the tree-sitter [4] parser generator. Approximately 2k lines of Python code was added to COMEX to support C/C++ syntax. The implementation of program slicing and new harness synthesis comprises around 3k lines of Python and shell scripts.

5.2 Experiment Setup

In this section, we present the evaluation of WildSync in the following aspects:

- **Automation & Design** How many new target functions can WildSync extract external usage for and synthesize harnesses? What level of human effort is still required in the process? (Section 5.3)
- **Effectiveness** What is the effectiveness of the new harnesses synthesized by WildSync in terms of increased test coverage and bug discovery? (Section 5.4)
- **Comparison** How does WildSync compare to other approaches for automatic harness generation? (Section 5.5)

All our experiments are conducted on cloud machines with 32-core Intel Xeon Gold 6142 CPUs at 2.6 GHz and 384GB of ECC DDR4-2666 memory provided by Cloudlab [10].

5.3 Synthesizing Fuzzing Harnesses with WildSync

To evaluate the effectiveness and practicality of WildSync, we widely selected 24 C/C++ projects from OSS-Fuzz to synthesize new harnesses. Firstly, as WildSync utilizes `ArchLinux`'s package manager, `Pacman`, to identify library dependencies, we found in total of 54 libraries on OSS-Fuzz with such matches. Next, we removed 20 libraries that were not suitable for harness synthesis based on the criteria discussed later in Section 6.4. After that, we focused on libraries with less than 70% existing test coverage on OSS-Fuzz, and exclude the libraries without functions to synthesize for.

Additionally, we synthesized harnesses for 3 libraries from `Pacman` that are not on OSS-Fuzz but actively maintained and widely used. These libraries are chosen for that they provide clear example code that we can easily create base harnesses with, and then we apply WildSync to synthesize more. In total, we include 27 open-source projects as target libraries to fuzz in our evaluation. It contains a wide category of libraries including compression, format parsing, media processing, etc.

The initial step of WildSync is to search for the exposed APIs and their external usage. WildSync fetched the source code of external projects depending on the target libraries, resulting in 304 projects retrieved for the 27 libraries. For each library, WildSync identified the exposed APIs and selected untested ones to synthesize harnesses with. This process selected 1174 APIs as potential targets among the 27 selected libraries.

Next, WildSync extracted the usage of the target functions from the external projects and synthesized new harnesses for them. Following the synthesizing process, WildSync automatically detected and removed invalid harnesses. As WildSync substitutes unknown variables and data types for integers and memory chunks, most of the invalid harnesses are eliminated for inconsistent type casting. After removing the non-compilable ones, WildSync successfully synthesized harnesses for 469 APIs. Each individual harness is created within a 30-second timeout, including the process from extracting target function usage, and data-flow analysis, to harness synthesis.

The process of synthesizing new harnesses is fully automated, with minimal human intervention required to incorporate the new harnesses into OSS-Fuzz's compile pipeline. Additionally, as most APIs are used in more than one external project, WildSync will be collecting them all and synthesizing multiple harnesses for one API. To effectively conduct the evaluation, we performed an additional manual review to select the most concise harness for each function. This review process required less than 1 hour in total for all 27 target libraries. In reality, the manual review process may not be necessary, as this can be further automated by implementing heuristics to keep all diverse harnesses to fully test the usage of the APIs, and exclude harnesses with exact duplicated usage.

The latter part of invalid harnesses deduction involves a try-fuzz of the harnesses, allowing developers to inspect harnesses that crashed immediately after fuzzing started. During this stage, all spurious crashes were identified, which we will discuss in Section 5.4.2. Additionally, 4 assertions were raised from `json-c` and 1 in `cairo` due to certain data structures not matching specific conditions when being passed to the target functions. These harnesses were manually fixed by adding checks before calling the target functions and re-entered into the pipeline for fuzzing.

Automation of WildSync: WildSync demonstrates its capability by automatically synthesizing new harnesses for 469 untested APIs spanning 27 libraries. This process demands minimal human intervention, primarily involving the addition of new harnesses to the existing OSS-Fuzz setup and the removal of redundant harnesses.

5.4 Effectiveness of WildSync

In the next section of the evaluation, we conducted fuzzing experiments with both the original OSS-Fuzz harnesses and the newly synthesized ones using libFuzzer. For the 3 additional libraries that are not on OSS-Fuzz, the original "OSS-Fuzz harness" column stands for the number of harnesses we manually constructed based on the example code provided by the library. These experiments ran for 24 hours across 5 rounds, using identical sets of initial seeds for harnesses under the same library. While WildSync 's synthesized harnesses are inherently compatible with most conventional fuzzers, we used libFuzzer due to its prominence in library testing. The outcome of the experiment is summarized in Table 2.

Table 2. Coverage increase of new harnesses synthesized with WildSync. The first part shows the coverage increase of the libraries already on OSS-Fuzz. The second **manual** section is the results of 3 libraries we started from stretch. We manually created initial harnesses from the example code provided by the developers, then expanded the API covered with WildSync. Each harness is fuzzed for 24 hours, 5 rounds with libFuzzer. **UC** = Total unique crashes, **S** = Spurious crashes, **N** = Normal crashes with timeout/out-of-memory, **B** = Crashes reported as bugs after manual review.

Target Library	# API covered		Line Coverage (avg)			Function Coverage (avg)			Crashes
	OSS-Fuzz	WildSync	OSS-Fuzz	WildSync		OSS-Fuzz	WildSync		#UC (S/N/B)
cairo	23	+56	12,366	+878	(+7.10%)	1,014	+97	(+9.58%)	0 (0,0,0)
cjson	6	+8	979	+183	(+18.69%)	31	+19	(+61.29%)	4 (0,4,0)
fribidi	3	+3	1,101	+144	(+13.08%)	20	+7	(+35.00%)	3 (0,3,0)
gdk-pixbuf	17	+14	1,593	+218	(+13.71%)	87	+13	(+14.71%)	8 (2,6,0)
json-c	4	+11	1,362	+544	(+39.94%)	68	+47	(+69.12%)	1 (0,1,0)
krb5	25	+5	6,561	+956	(+14.57%)	550	+64	(+11.64%)	1 (0,1,0)
lcms	67	+26	10,550	+400	(+3.79%)	714	+15	(+2.04%)	9 (0,8,1)
leptonica	278	+9	27,363	+897	(+3.28%)	1,168	+44	(+3.77%)	7 (1,6,0)
libarchive	33	+8	14,688	+1787	(+12.17%)	696	+44	(+6.32%)	7 (0,7,0)
libass	13	+4	6,415	+113	(+1.76%)	371	+8	(+2.10%)	4 (3,1,0)
libgd	8	+6	1,658	+62	(+3.75%)	79	+9	(+11.14%)	6 (0,6,0)
libheif	40	+3	4,200	+1366	(+32.53%)	338	+139	(+41.15%)	0 (0,0,0)
libpcap	9	+11	7,776	+1094	(+14.07%)	242	+40	(+16.52%)	3(0,3,0)
libplacebo	27	+4	2,258	+125	(+5.53%)	197	+12	(+5.88%)	2 (0,0,2)
libpng	21	+28	5,171	+381	(+7.37%)	161	+29	(+18.16%)	1 (0,1,0)
libsoup3	7	+14	99	+414	(+416.70%)	11	+32	(+287.27%)	4 (3,0,1)
libtiff	8	+14	12,981	+235	(+1.81%)	402	+21	(+5.22%)	14 (0,14,0)
libvnc	4	+13	871	+650	(+74.57%)	54	+29	(+54.37%)	16 (2,13,1)
libyang	11	+72	11,260	+3,047	(+27.06%)	469	+162	(+34.54%)	7 (6,1,0)
lmdb	13	+13	2,065	+341	(+16.51%)	78	+17	(+21.23%)	0 (0,0,0)
mxml	2	+12	1,087	+203	(+18.71%)	39	+11	(+28.57%)	13 (1,12,0)
vorbis	3	+6	2,463	+44	(+1.77%)	96	+7	(+7.29%)	0 (0,0,0)
yyjson	6	+29	3,795	+341	(+8.98%)	128	+235	(+183.59%)	2 (1,1,0)
zlib	29	+22	3,350	+209	(+6.23%)	94	+20	(+21.06%)	2 (1,1,0)
total	657	+391	142,013	+14,632	(+10.30%)	7,107	1,119	(+15.74%)	114 (20,89,5)
	manual	WildSync	manual	WildSync		manual	WildSync		
libcue	2	+4	251	+78	(+31.08%)	31	+6	(+19.35%)	1 (0,0,1)
libharu	33	+40	1,845	+604	(+32.72%)	137	+40	(+29.20%)	19 (8,11,0)
libxmlb	15	+34	1,423	+1623	(+114.07%)	116	+136	(+117.47%)	32 (2,27,3)
total	50	+78	3,519	+2,305	(+65.50%)	284	+182	(+64.10%)	52(10,38,4)
All total	707	+469	145,532	+16,937	(+11.64%)	7,391	+1,301	(+17.60%)	166 (30,127,9)

5.4.1 Coverage Improvement. From Table 2, substantial increases in both line and function coverage can be observed across various libraries. For libraries that have already undergone extensive fuzzing, such as `leptonica` having as many as 45 harnesses, the absolute coverage improvements remain remarkable.

However, it's important to note that the number of newly covered functions doesn't always directly correspond to the addition of new APIs. For example, in the case of `libpng`, the selected

new APIs primarily consist of shallow functions with limited additional function calls, resulting in the coverage of only 29 new functions across 28 newly synthesized harnesses. In `vorbis`, the low ratio of new APIs to coverage is due to the need for different options to be set during initialization, a feature not yet supported by the current implementation of WildSync.

Regarding `gdk-pixbuf`, some of the new harnesses are derived from existing ones that are not fully covered in the current OSS-Fuzz setup. Addressing bottlenecks in the original harnesses could unlock further coverage potential. In contrast, despite introducing only 3 new API entries in `libheif`, the synthesized harnesses achieve a substantial 139 new function coverage. This highlights the efficacy and potential of WildSync in enhancing test coverage and identifying unexplored areas in software libraries.

Coverage Improvement of WildSync: Based on the existing harnesses, WildSync successfully synthesized new harnesses for 469 APIs across 27 libraries. This led to the addition of nearly 20k lines of code and coverage of over 1.3k new functions, significantly boosting the testing capabilities of the libraries under test.

5.4.2 Bug Hunting. In the evaluation of WildSync, we encountered in total of 166 errors triggered by libFuzzer, which can categorized into three sets: bugs discovered after manual review, spurious crashes due to API misuse, and timeout/out-of-memory.

First, after manually reviewing the crashes, we discovered 7 new bugs with the synthesized harnesses. The numbers and types of bugs triggered are listed in Table 3. At the time of submitting this paper, we have reported this issue to the original developers of the libraries, and 5 of the bugs are confirmed and fixed.

Table 3. Reported bugs found by WildSync. Report ID **I**=issue, **P**=pull request. **Patched**=patch submitted but not yet merged.

Library	API entry	Bug Type	Status	Report ID
lcms	cmsDupContext	Null pointer dereference	Fixed	462 (I)
libplacebo	pl_options_load	Out-of-bound array access	Fixed	326 (I)
	pl_shader_finalize	Assertion failed	Fixed	327 (I)
libsoup3	soup_message_headers_get_content_type	SEGV on unknown address	Fixed	389 (I)
libvnc	rfbRunEventLoop	Buffer overflow	Reported	615 (I)
libcue	cd_get_track	SEGV on unknown address	Patched	60 (P)
libxmlb	xb_silo_query xb_silo_query_build_index xb_silo_query_first	SEGV on unknown address	Fixed	208 (I)

As WildSync extracts external code to synthesize new harnesses, it is expected to collect the external code initializing and applying sanitize checks of the function arguments before calling the target API. For example, some APIs receiving a pointer as a function parameter may require the pointer to be initialized and non-null when passing to the function, and such checks are often left to the API users. If a synthesized harness crashes for failing to provide such checks, it should be considered a bug in the external project. However, the could also be some cases which the library maintainers willing to fix the issue for the better safety of the API usage. In Figure 5 we showed a bug discovered by WildSync. Originally `libsoup` left the sanitize check to external developers, as done in Figure 5 (a) line 2. However, WildSync extracted the usage of `soup_message_headers_get_content_type` from various external code, and discovered that all implementations will directly pass the return from another function, `soup_message_get_response_headers`, which could be a null pointer, leading to a crash in the API. So, before reporting to all libraries using `libsoup`, we successfully convinced the library maintainer that a sanitizer check should be deployed internally to ensure the safety of the API usage.

Among the errors triggered by harnesses generated with WildSync, only one was due to WildSync failed to collect the API sanitizing check for that the check being out of reach of the current implementation. We categorized this as a spurious crash. All the necessary checks, if any, are correctly collected in other harnesses.

5.4.3 Spurious Crashes & Other Errors. During the try-fuzz stage, 29 crashes are identified early and categorized as spurious crashes. Among them, 18 of these crashes are due to the current design of WildSync attempting to assign random memory chunks to library specific data structures that are not initialized. Unfortunately, these data structures often contain a hierarchy of pointers, which will crash a harness immediately when being accessed and dereferenced. This can be improved with future work on extracting multiple code snippets to initialize these data structures smartly. 3 errors are due to the APIs expecting a data array along with a size parameter in `libass`. A program will easily crash if the size is not correctly set up when using these APIs. Although this could be leveraged as a potential bug in the program utilizing `libass` and we had reported this to the developers of the external projects, we categorized them as spurious bugs before concrete exploits were discovered.

```
SoupMessage *message = soup_message_new (SOUP_METHOD_GET, use_uri);
if (!message) goto cleanup;
/* .... */
const char* header_type = soup_message_headers_get_content_type(soup_message_get_response_headers (message), NULL);
```

(a) One of the external usage of `soup_message_headers_get_content_type`.

```
const char * soup_message_headers_get_content_type (SoupMessageHeaders *hdrs, GHashTable **params) {
+   g_return_val_if_fail (hdrs, NULL);
    if (!hdrs->content_type) return NULL;
    /* .... */ }
```

(b) Sanitize check being added to `soup_message_headers_get_content_type` and all other relative functions after the patch.

Fig. 5. Reported bug in `libsoup` and the corresponding patch.

The remaining 9 spurious crashes consist of various reasons, such as falsely fuzzing API arguments taking file paths with randomly generated strings. Or a synthesized harness feeding fuzzing inputs to an API, which are constant values, but the API attempts to write to the function arguments and results in a crash. One is due to WildSync's failure to collect an API sanitizing check as described in Section 5.4.2.

Lastly, 127 errors triggered are libFuzzer aborts due to timeout or out-of-memory (a single input taking more than 25 seconds or 2.5GB RAM to process). While these crashes could be potential issues such as infinite loops, most of them may not necessarily be considered urgent issues to be resolved by the library maintainers. Plus, plenty of the timeout or out-of-memory issues exist in the original harnesses from which WildSync synthesizes from. We believe that most of these errors should be resolved when the bottlenecks in the original harness are fixed.

Bug Hunting of WildSync: With the newly synthesized harnesses, WildSync discovered 7 new bugs across the 27 open-source libraries. These functions remained untested by OSS-Fuzz despite the extensive fuzzing efforts on these libraries; however, our newly developed harnesses successfully triggered crashes.

5.5 Comparison with Existing Works

In this section, we mainly compare WildSync with Hopper [8], a library API fuzzing engine that infers intra- and inter-API constraints *within* libraries and mutates programs with grammar awareness. This design contrasts with WildSync, which derives such constraints *outside* the

libraries. Hopper automatically learns API constraints and data dependencies within a library, then generates harnesses by mutating API sequences and values passing as function arguments. In its paper evaluation, Hopper outperforms the test coverage of OSS-Fuzz and another automatic harness synthesizer, GraphFuzz [11], which also applies internal library analysis to perform API testing, discovering several new bugs.

To fairly evaluate WildSync's effectiveness, we analyzed the same set of libraries listed in Table 2 using Hopper, with results presented in Table 4. Additionally, to ensure a thorough comparison, we applied WildSync to the library versions evaluated in Hopper's original study and compare the bugs discovered by both tools. The results can be found in Table 5.

Another related and publicly accessible tool is FuzzGen [12], which performs whole-system analysis to infer valid API interactions for a target library and synthesize fuzzing harnesses. While conceptually similar to WildSync in its goal of extracting API usage from real-world scenarios, FuzzGen primarily targets Android libraries and provides undocumented support for Linux Debian system. Additionally, FuzzGen requires recompiling all external codebases with LLVM bitcode to extract API usage, making it challenging to apply universally to compare with WildSync, as the external projects WildSync analyzes lack a common build system. In Hopper's paper, the authors were only able to successfully run FuzzGen on one library, and instead referenced results for `libvpx` and `libaom` from FuzzGen's original paper. For consistency, we also include these numbers in our evaluation, as shown in Table 5.

Table 4. Comparison of WildSync with Hopper. **S** = Spurious crashes, **B** = Crashes reported as bugs after manual review. (Timeout and memory errors are excluded from the comparison for they being libFuzzer specific features.)

Library	APIs covered w/o spurious			Line Coverage (avg)		Function Coverage (avg)		Line / API		Unique Crashes (S/B)	
	Hopper	WildSync	∩	Hopper	WildSync	Hopper	WildSync	Hopper	WildSync	Hopper	WildSync
cairo	328	78	73	18,552	13,244	1,552	1,112	56.56	169.79	16 / 0	0 / 0
cjson	78	15	15	2,103	1,162	112	50	26.96	77.47	1 / 1	0 / 0
fribidi	32	7	7	2,272	1,245	78	27	71.01	177.86	6 / 0	0 / 0
json-c	131	18	18	3,206	1,906	207	115	24.47	105.89	60 / 1	0 / 0
krb5	39	7	0	1,057	7,517	88	614	27.10	1,073.89	2 / 0	0 / 0
lcms	280	56	51	12,166	10,951	866	729	43.45	195.56	123 / 0	1 / 1
libarchive	265	40	18	12,327	16,475	868	740	46.52	411.88	41 / 0	0 / 0
libass	45	12	11	1,571	6,529	111	379	34.92	544.05	19 / 0	3 / 0
libgd	229	15	10	5,124	1,720	298	88	22.38	114.65	22 / 1	0 / 0
libheif	109	31	16	2,738	5,567	392	477	25.12	179.57	71 / 0	0 / 0
libpcap	82	20	20	6,801	8,870	323	282	82.94	443.51	39 / 0	0 / 0
libpng	194	54	53	4,917	5,552	298	190	25.35	102.81	15 / 0	0 / 0
libtiff	169	20	18	9,820	13,215	409	423	58.11	660.76	36 / 0	0 / 0
libvnc	21	15	0	453	1,521	32	81	21.59	101.41	0 / 0	2 / 1
libyang	266	48	39	13,312	14,308	736	631	50.04	298.08	57 / 0	6 / 0
lmdb	50	26	22	836	2,406	67	95	16.72	92.52	11 / 0	0 / 0
mxml	91	15	15	1,388	1,290	95	50	15.25	86.03	15 / 0	1 / 0
vorbis	34	9	0	696	2,507	44	103	20.48	278.51	77 / 0	0 / 0
yyjson	43	37	3	4,679	4,136	199	363	108.82	111.77	34 / 0	1 / 0
zlib	81	37	36	3,779	3,559	143	114	46.66	96.18	5 / 0	1 / 0
libcue	9	5	4	886	329	36	37	98.49	65.80	0 / 0	0 / 1
libharu	472	37	30	5,575	2,449	534	177	11.81	66.18	34 / 0	8 / 0
Total	2,807	602	423	114,260	126,456	7,489	6,877	40.71	210.06	684 / 3	22 / 3

5.5.1 Comparison with Hopper. Table 4 presents a comparison between WildSync and Hopper. Five libraries were excluded from the evaluation due to errors Hopper encountered while resolving certain header files. For each remaining library, we conducted 5 independent 24-hour trial runs with Hopper. In the table, we present the number of new APIs successfully covered (i.e. excluded the ones leading to spurious crashes), average line and function coverage, and the number of bugs discovered by both tools.

Hopper is designed to generate programs that broadly cover a library's APIs without relying on any pre-existing knowledge. It successfully achieves high API coverage across the libraries we

evaluated. In some cases, this extensive API coverage indeed leads to high line coverage. Hopper excels in these libraries due to two key factors. First, it covers multiple entry-point APIs from various sources, such as reading from memory or files, and attempts to initialize additional data structures that are not necessarily handled by existing OSS-Fuzz harnesses. Secondly, its harnesses include utility functions (e.g. library customized math functions) that, while not directly related to a library's core functionality, still contribute to broader line coverage. However, for half of the evaluated targets, WildSync achieves significantly higher line coverage despite utilizing harnesses that cover fewer APIs. This highlights the importance of well-tuned setup harnesses, which remain crucial for effective fuzzing in many cases. Simply covering a large number of APIs does not necessarily translate to better fuzzing efficiency if the harnesses are not optimized for deeper code execution.

Also, Hopper generates a significant number of false positives in the form of crashes, primarily caused by API misuse. Since Hopper learns API constraints while performing fuzzing and harness generation, it does not inherently guarantee that the generated API sequences are valid. As a result, many crashes occur due to incorrect API usage rather than actual bugs in the target libraries.

The original number of crashes Hopper produces is as high as 21,122 crashes. We first apply Hopper's learned constraints as a first filter to reduce this number to 10,741 crashes for further debugging. To manage this volume, we leverage CASR [24] to cluster crashes with similar stack traces, which still result in 687 unique groups of crashes to investigate. After manually reviewing these cases, we found only three crashes were deemed valid issues worth reporting, while the rest were classified as spurious. Among these bugs, we reported the bug Hopper found in `libgd`, which has been acknowledged by the library maintainers and is pending a fix. For the other two, the one in `cjson` was previously reported by Hopper's authors but remains unfixed in the new library version we tested on. The other in `json-c` is classified as a "document bug" by the library maintainers and has no planned mitigation[2].

The high number of spurious crashes generated by Hopper can be attributed to two major factors. The first is its attempt to synthesize non-primitive data types for API calls. Although Hopper analyzes the internal implementation of target libraries and tries to construct hierarchical structures for these data types, this still often results in invalid test cases that lead to crashes without meaningfully testing the library. The second major cause is the generation of invalid API call sequences. We observed multiple crashes caused by harnesses accessing data structures after calling APIs that free those structures. In contrast, WildSync mitigates such issues by categorizing API calls into groups to enforce a correct execution order. The large volume of spurious crashes raises questions about the extent to which Hopper's high line coverage reflects meaningful testing. While extensive coverage can be beneficial, its effectiveness in uncovering real bugs may be limited if it primarily generates invalid crashes rather than facilitating deep exploration of a library's functionality. The distinct coverage patterns and bug discovery of WildSync and Hopper also indicate the usefulness of WildSync's design. Although WildSync may not achieve the same level of API coverage as it relies on high-quality existing harnesses and external usage, it still opens up new areas to explore a library's functionality more effectively and to discover real bugs with fewer spurious crashes.

Coverage Comparison with Hopper: Hopper achieves high API coverage and results in extensive line coverage in some cases. Meanwhile, WildSync attains higher line coverage with fewer APIs, and further extends test coverage with less spurious crashes. Each design has its own strengths, balancing coverage breadth and testing effectiveness.

[2]https://github.com/json-c/json-c/issues/881

Besides the evaluation on our collected benchmarks, we apply WildSync to the versions of libraries presented in Hopper's paper, and compare the coverage improvements achieved and bugs discovered by both tools. We use 10 out of 11 libraries from Hopper's evaluation, and present the results in Table 5. The `re2` library is excluded as it is mainly used in its C++ API, and is currently out-of-scope for WildSync. The authors of Hopper attempted to reproduce FuzzGen's results on the same set of libraries, but could only successfully generate harnesses for `cJSON`. We also include this number in the table, along with the results of `libvpx` and `libaom` presented in FuzzGen's original paper.

Table 5. Compairson with other tools on previous versions of libraries. The coverage and bug information are referenced from the resuls presented Hopper's paper [8], in which FuzzGen failed to synthesize harnesses for most of the libraries.

Libraries (commit)		Line Coverage			Bug		
		FuzzGen	Hopper	WildSync	Function (Issue #)	Hopper	WildSync
cJSON	(b45f48e)	186	1,997	1,157	cJSON_DetachItemViaPointer (#722)	✓	x No external usage
					cJSON_ReplaceItemViaPointer (#726)	✓	x No external usage
c-ares	(4b7301a)	-	5,012	2,730	ares_set_sortlist (#496)	✓	x No external usage
libpng	(f1848a3)	-	9,610	5,559	png_warning (#453)	✓	✓
					png_image_write_to_file (#489)	API mis-use	-
lcms	(ef7bd0d)	-	9,001	10,834	cmsIsCLUT (#350)	✓	x No external usage
					cmsBuildTabulatedToneCurveFloat (#351)	✓	x Bug free with external usage
					cmsGetPostScriptCRD (#353)	✓	x Bug free with external usage
					cmsIT8SaveToMem (#354)	✓	x Bug free with external usage
					cmsIT8GetProperty (#355)	x	✓
libpcap	(acc5cb9)	-	7,536	7,984	pcap_breakloop (#1147)	✓	✓
					pcap_activate (#1098)	✓	✓
sqlite3	(dcd7408)	-	25,356	38,844	sqlite3_overload_function (bbbbb66b6b)	✓	✓
zlib	(e554695)	-	3,502	3,558	gzsetparams (#761)	API mis-use	-
					gzungetc (#837)	✓	✓
					gzflush (#840)	✓	✓
libmagic	(0fa2c8c)	-	4,230	5,544	-		
libvpx	(87315c0)	15,211	15,641	19,633	-		
libaom	(9a83c6a)	32,576	36,218	42,640	-		

First, the coverage comparison between WildSync and Hopper is similar to the previous evaluation. Hopper can achieve higher test coverage by successfully covering more APIs in some libraries, while WildSync performs better in others. Particularly, in `libpng`, Hopper's developers specifically configured the starting APIs to cover more library-specific data structures, while WildSync only has the existing OSS-Fuzz harness that covers less. We believe that if given the same amount of entry APIs, WildSync can achieve comparable coverage with Hopper.

For the vulnerability discovery, we list out the bugs reported by Hopper that have been fixed or confirmed by the library maintainers. Note that there are 2 bugs Hopper claimed in `zlib` and `libpng` that require certain sequences of API calls to trigger the crashes. The library maintainers actually denied to recognize these as issues, stating them being per library user's responsibility to ensure the correct usage. With WildSync's approach, such rejection is more confidently avoided as the usages of APIs are extracted from real-world scenarios. It will be either the interest of target library maintainers to fix the bug as the case in Section 5.4.2, or a mis-use that we report to external projects.

For the other 13 verified bugs reported by Hopper, WildSync is able to trigger 6 of them, while discovering an additional bug missed by Hopper that is later reported by other. For the bugs not triggered by WildSync, 4 of them are due to no external usage found. The other 3 are all issues related to passing unchecked NULL pointers or value 0 to the APIs. These APIs are covered by WildSync, but no crash is being triggered for that the real-world usage we collected is all bug-free. We believe that WildSync will be able to capture these issues once observing any potential misbehavior usage of the APIs in the external projects.

Bug-Hunting Comparison with Hopper: The different behavior between Hopper and WildSync in bug discovery reflects the diverse design of the two tools. We believe neither approach is superior to the other, and the two can be combined to provide a more comprehensive test coverage for libraries.

5.5.2 Comparison with Other Tools. Other prior works highly related to WildSync, such as Fudge [6], IntelliGen [28], and Daisy [29], are not publicly available. As a consequence, comparing WildSync directly with these works poses a challenge. However, we propose that the experiments conducted in this paper, particularly the comparison with existing, long-tested libraries on OSS-Fuzz, offer a comprehensive approach to evaluate WildSync's performance relative to these works. Our rationale stems from two key observations. First, Fudge reported that the harnesses generated by their tool were accepted and hosted on OSS-Fuzz, notably for the `leptonica` library. As depicted in Table 2, `leptonica` boasts an exceptionally high number of existing harnesses. Despite this, WildSync was able to synthesize 9 new harnesses for this library, covering nearly 900 new lines and 44 new functions. This underscores the efficacy of WildSync in augmenting existing harnesses and identifying areas for further coverage. Second, at the time we conducted our evaluation, OSS-Fuzz achieves significantly higher coverage than in the reported results of IntelliGen and Daisy. This highlights the continuous growth and improvement of OSS-Fuzz over time, positioning it as a more up-to-date benchmark for evaluating the effectiveness of WildSync.

Another recent work related to fuzzing harness synthesis is UTopia [15], which aims to generate harnesses from existing unit tests with minimal human involvement. Unfortunately, the released implementation of UTopia supports only a limited number of C++ unit-test frameworks, and is not applicable to the benchmarks we use for WildSync's evaluation.

In summary, while direct comparison with more prior works may be challenging due to their unavailability or different implementation restrictions, the experiments conducted in this paper offer a robust evaluation framework for assessing WildSync's capabilities in enhancing fuzz testing for software libraries.

6 Discussion

6.1 WildSync's Time Efficiency

For the current implementation of WildSync, we cap our harness synthesis process with a 30-second timeout for each API. The decision comes from the fact that the algorithm is only extracting one API usage from one single source file at a time, and should not take long. WildSync first locates files containing the target API usage in the source code of external projects. Then, it parses the AST of these located files to identify the relevant external functions using the target API. In this phase, most of the files can be parsed within 5 second, except for some large files with more than 20k lines of code. After that, WildSync constructs CFG and DFG individually for these external functions to perform program slicing. The current implementation can handle a single functions with up to 1k lines of code or cyclomatic complexity lower than 50.

6.2 Combining with Other Techniques

The core concept behind WildSync is to gather how a target library is utilized in the wild and leverage the knowledge to generate fuzzing harnesses effectively. We envision that by integrating WildSync with other techniques that focus on analyzing API dependencies internally, we can create more comprehensive harnesses. This integration can help highlight the knowledge gap between the internal library developers and external users.

Furthermore, we believe that the approach employed by WildSync can be extended to other testing techniques beyond fuzzing, such as unit-test case generation. By leveraging external usage

patterns, developers can gain valuable insights into potential edge cases and usage scenarios to enhance their testing infrastructure. This broader application of WildSync has the potential to enhance testing practices across various domains and improve software reliability and robustness.

6.3 Extending to Other Languages

While WildSync currently focuses on generating harnesses for C libraries, we believe that its underlying ideas can be extended to support other programming languages. However, this expansion would require a thorough analysis of language-specific features and constructs. For instance, when attempting to apply the current version of WildSync to libraries written purely in C++, we encountered challenges related to namespace resolution and the object-oriented nature of API calls. Addressing these issues would involve adapting WildSync's techniques to handle C++'s unique characteristics, such as classes, inheritance, and polymorphism. Exploring the extension of WildSync to other languages could significantly broaden its applicability and impact, enabling developers to automatically generate fuzzing harnesses for a wider range of software projects. This extension would involve incorporating language-specific parsing and analysis techniques to effectively capture usage patterns and generate robust harnesses.

6.4 Libraries Applicable with WildSync

Despite its promising and extensible nature, WildSync may face challenges when dealing with system-level libraries that handle file system or network operations. These libraries often require specific environmental setups to be tested accurately and may have limited interfaces suitable for fuzzing. Cryptography libraries are excluded as well for they requiring standard and strict call sequences. As a result, WildSync's ability to generate effective fuzzing harnesses for such libraries may be limited at the moment. However, we believe that by integrating WildSync with other techniques that focus on system-level testing, we can overcome these challenges and extend its applicability to a broader range of libraries and software projects.

6.5 Large Language Model Based Fuzz Driver Generation

Fuzzing APIs is inherently challenging, and recent work has explored leveraging LLM-based (Large Language Model) approaches to automate fuzzing harness generation. Google's OSS-Fuzz-Gen [16] represents one of the first attempts in this direction, demonstrating the feasibility of using LLMs for this task. However, the generalizability of such approaches remains uncertain. Zhang et al. [27] conducted an in-depth study investigating key challenges in using LLMs for effective fuzz driver generation, offering critical insights into their limitations and potential improvements. In parallel, WildSync provides a deterministic program analysis approach to automate harness synthesis. We believe that WildSync will foster future research in this area, potentially providing a hybrid alternative or design that enhances the robustness and adaptability of automated fuzzing harness generation.

7 Related Work

7.1 Coverage-based Fuzzing

AFL, libfuzzer, honggfuzz [21, 23, 25] are the three state-of-the-art coverage-based fuzzer. They have proved to be high-performance and pragmatic fuzzing engines by being deployed in numerous projects. They have also inspired countless research projects. Angora [7] is a Rust-based fuzzer that tried to collect more information than AFL but without losing efficiency. In order to mutate input efficiently, Angora incorporates taint analysis [20] to track which parts of the program are affected by input values. It also introduced several novel techniques including byte-level taint tracking,

context-sensitive branch count, gradient descent search, and input length exploration. Redqueen [5] introduced the concept of "Input-to-State Correspondence". The authors observed that often there are direct relationships between input and program states. They introduced 'redqueen', a lightweight instrumentation to track these relationships alternative to taint analysis.

7.2 Automatic Harness Synthesis

Library fuzzing is a challenging task as it requires constructing a fuzzing harness that contains the desired sequences of library API calls to test with. However, as the number of libraries grows, it becomes increasingly difficult to manually create fuzzing harnesses for each library. Several works have been presented to generate fuzzing harnesses automatically. FuzzBuilder [13] and FuzzBuilderEx [14] aim to automate the transformation of unit test cases into fuzzing harnesses, but they require manual configuration and the resulting harnesses may still require significant manual effort to ensure quality. Fudge [6] automatically generates fuzzing harness candidates based on buffer access parameter signatures from usage extracted from client code, but human intervention is needed to evaluate and update the generated code. FuzzGen [12] leverages whole-system analysis to infer API dependencies and generate fuzzers, but manual review is required to repair the generated programs. IntelliGen [28] focuses on identifying entry functions and generating fuzzing harnesses, but it may overlook API relations, leading to incomplete harnesses. Daisy [29] models object behaviors and constructs interface calls based on object usage sequences, while GraphFuzz [11] and RUBICK [26] use graph-based techniques to model API usage and control dependencies. These approaches show promise in modeling object-oriented API usage but may require sophisticated techniques based on compilers or runtime execution, limiting their scalability without manual intervention. In contrast, WildSync performs analysis on abstract syntax trees obtained from syntax highlighters, requiring no additional human effort beyond the presence of code snippets. While it may not capture complete API call sequences, it offers a scalable solution for synthesizing fuzzing harnesses.

Recent work UTopia [15] automates the synthesis of fuzzing harnesses from existing unit tests, but it is currently limited to certain build systems. That being said, UTopia is a promising work that can be integrated with WildSync to further improve the quality of the synthesized harnesses. Hopper [8] tries to learn the API usage pattern from the source code to generate fuzzing harnesses. As demonstrated in the evaluation, as being able to covering more APIs as being a bottom-up appraoch, its harnesses' mutated API sequences may be too distinct from normal usage.

8 Conclusion

In this paper, we presented WildSync, a novel approach to automatically synthesize fuzzing harnesses for open-source libraries. WildSync is designed to be light-weight, scalable, and with limited manual intervention to generate harnesses for previously untested functions. We demonstrated the effectiveness of WildSync by applying it to 27 C/C++ libraries. As a result, WildSync is able to generate 469 new harnesses for these libraries. With the new harnesses, we observed up to 400% test coverage improvement for these libraries, and identified 7 new bugs after manual review. Overall, these findings underscore the significant potential of WildSync to enhance software testing and improve the reliability of open-source libraries, paving the way for future research into further automation and integration of fuzzing techniques in diverse software ecosystems.

9 Data Availability

We make WildSync publicly available at: https://github.com/spencerwuwu/WildSync along with scripts to reproduce the results presented in the paper.

References

[1] 2021. Cisco secure development lifecycle. https://www.cisco.com/c/dam/en_us/about/doing_business/trust-center/docs/cisco-secure-development-lifecycle.pdf

[2] 2021. Microsoft Security Development Lifecycle. https://www.microsoft.com/en-us/securityengineering/sdl/practices

[3] 2024. Fuzzing Introspection of OSS-Fuzz projects. https://introspector.oss-fuzz.com/projects-overview

[4] 2024. Tree-sitter. https://tree-sitter.github.io/tree-sitter/

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2018. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Network and Distributed System Security Symposium (NDSS)*.

[6] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, 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*. 975–985.

[7] Peng Chen and Hao Chen. 2018. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (Oakland)*.

[8] 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.

[9] Debeshee Das, Noble Saji Mathews, Alex Mathai, Srikanth Tamilselvam, Kranthi Sedamaki, Sridhar Chimalakonda, and Atul Kumar. 2023. COMEX: A Tool for Generating Customized Source Code Representations. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2054–2057. doi:10.1109/ASE56229.2023.00010

[10] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[11] 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*. 1070–1081.

[12] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287.

[13] Joonun Jang and Huy Kang Kim. 2019. Fuzzbuilder: automated building greybox fuzzing environment for c/c++ library. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 627–637.

[14] Sanghoon Jeon, Minsoo Ryu, Dongyoung Kim, and Huy Kang Kim. 2022. Automatically Seed Corpus and Fuzzing Executables Generation Using Test Framework. *IEEE Access* 10 (2022), 90408–90428.

[15] 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)*. IEEE, 2676–2692.

[16] Dongge Liu, Oliver Chang, Jonathan metzman, Martin Sablotny, and Mihai Maruseac. 2024. *OSS-Fuzz-Gen: Automated Fuzz Target Generation*. https://github.com/google/oss-fuzz-gen

[17] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[18] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[19] Max Moroz and Kostya Serebryany. 2016. Guided in-process fuzzing of Chrome components. *Google Security Blog* (2016).

[20] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy (Oakland)*.

[21] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *IEEE Cybersecurity Development Conference (SecDev)*.

[22] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security Symposium (USENIX)*.

[23] Robert Swiecki. 2018. honggfuzz. http://honggfuzz.com/

[24] Ilya Yegorov and Georgy Savidov. 2024. Crash Report Accumulation During Continuous Fuzzing. In *Ivannikov Memorial Workshop 2024*. IEEE. https://arxiv.org/abs/2405.18174

[25] Michal Zalewski. 2017. American fuzzy lop. http://lcamtuf.coredump.cx/afl/

[26] 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 *Proceedings of the 32nd USENIX Conference on Security Symposium*. 2867–2884.

[27] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1223–1235. doi:10.1145/3650212.3680355

[28] 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.

[29] 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)*. IEEE, 87–98.

Received 2024-10-30; accepted 2025-03-31