No Harness, No Problem: Oracle-guided Harnessing for Auto-generating C API Fuzzing Harnesses

Gabriel Sherman
University of Utah

Stefan Nagy
University of Utah

***Abstract*—Library APIs are used by virtually every modern application and system, making them among today's most security-critical software. In recent years, library bug-finding efforts have overwhelmingly adopted the powerful testing strategy of coverage-guided fuzzing. At its core, API fuzzing operates on *harnesses*: wrapper programs that initialize an API before feeding random inputs to its functions. Successful fuzzing demands *correct* and *thorough* harnesses, making manual harnessing challenging without sufficient domain expertise. To overcome this, recent strategies propose "learning" libraries' intended usage to automatically generate their fuzzing harnesses. Yet, despite their high code coverage, resulting harnesses frequently *miss* key API semantics—bringing with them invalid, unrealistic, or otherwise-impossible data and call sequences—derailing fuzzing with false-positive crashes. Thus, without a precise, *semantically-correct* harnessing, many critical APIs will remain beyond fuzzing's reach—leaving their hidden vulnerabilities ripe for attackers.**

This paper introduces *Oracle-guided Harnessing*: a technique for fully-automatic, semantics-aware API fuzzing harness synthesis. At a high level, Oracle-guided Harnessing mimics the trial-and-error process of *manual* harness creation—yet automates it via fuzzing. Specifically, we leverage information from API headers to mutationally stitch-together candidate harnesses; and evaluate their validity via a set of *Correctness Oracles*: compilation, execution, and changes in coverage. By keeping—and further mutating—only correct candidates, our approach produces a diverse set of semantically-correct harnesses for complex, real-world libraries in as little as one hour.

We integrate Oracle-guided Harnessing as a prototype, OGHARN; and evaluate it alongside today's leading fully-automatic harnessing approach, Hopper, and a plethora of developer-written harnesses from OSS-Fuzz. Across 20 real-world APIs, OGHARN outperforms developer-written harnesses by a median 14% code coverage, while uncovering 31 and 30 more vulnerabilities than both Hopper and developer-written harnesses, respectively—with zero false-positive crashes. Of the 41 *new* vulnerabilities found by OGHARN, all 41 are confirmed by developers—40 of which are since fixed—with many found in APIs that, until now, lacked harnesses whatsoever.

I. INTRODUCTION

Coverage-guided fuzzing remains one of today's most successful approaches for software vulnerability discovery. Much of real-world fuzzing targets *library APIs*, as they form the fundamental building blocks of today's critical applications and systems. Today, thousands of APIs are fuzzed continuously via Google's OSS-Fuzz [69]: the largest API fuzzing effort ever undertaken. In recent years, academic and industrial enhancements continue to improve API fuzzing speed [68], [89], practicality [27], [21], and effectiveness [14], [60].

Fuzzing APIs requires *harnesses* (or "drivers"): wrapper applications that instantiate the library and its objects before feeding fuzzer-generated inputs to its targeted function(s). Yet, writing harnesses is no easy task. Subtle mistakes in library initialization [7], data and arguments [8], and call sequences [6] often derail fuzzing with hundreds of false-positive crashes. Preventing such errors demands intimate knowledge of a library's semantics—often possessed by its developers alone—making manual harnessing unscalable for general-purpose library fuzzing. To this end, recent *automated* fuzzing harness generators are extending fuzzing's reach across the broader API ecosystem [54], [53], [20].

At a high level, automated harness generators aim to "learn" a library's semantics to ensure their resulting harnesses are correct. Previous-generation harnessers infer API semantics by statically analyzing available "reference" code: unit tests [54], applications [91], [53], and even other harnesses [12]. However, reference-based harnessing falls short when reference code is absent, low quality, or minimal in its use of API functionality. To remedy this, Chen et al. [20] propose Hopper: a strategy for *dynamic* harness generation. Rather than rely on reference code, Hopper performs on-the-fly harnessing by mutating API function calls and data. Yet, despite Hopper's ability to harness many API functions, its harnesses frequently violate key library semantics—valid argument ranges, call ordering, and data initialization—creating numerous false-positive bugs solely from APIs' *misuse*. Thus, without a correct, semantics-*aware* harnessing strategy, critical API vulnerabilities will remain undiscovered by today's state-of-the-art fuzzers—leaving dangerous opportunities for attacker exploitation.

To overcome this challenge, we propose *Oracle-guided Harnessing*: an automated strategy for generating semantically-valid library fuzzing harnesses—entirely from scratch. At its core, Oracle-guided Harnessing operates like a fuzzer—iteratively mutating harnesses, while filtering-out semantically-invalid ones via a suite of *Correctness Oracles*: harness compilation, execution, and code coverage. Unlike prior harnessers that require extensive reference code [53], [12], [91], Oracle-guided Harnessing needs only a library's headers to begin synthesizing valid harnesses; and unlike current on-the-fly harnessing [20], [48], it *upholds* critical API semantics whose violations derail fuzzing with false-positive crashes. When paired with a fuzzer [26], Oracle-guided Harnessing enables greater coverage—and bug discovery—while extending fuzzing's reach to many more libraries than previously possible.

We implement Oracle-guided Harnessing as a prototype, OGHARN, and evaluate it on **20 real-world libraries** alongside today's state-of-the-art harnesser Hopper [20] and

33 developer-written OSS-Fuzz [69] harnesses. Across five 24-hour fuzzing campaigns per benchmark, OGHARN outperforms manually-written harnesses by a median **14%** code coverage; while uncovering **31** and **30** more security bugs than both, respectively. We further show Oracle-guided Harnessing is unsurpassed in correctness: compared to Hopper's false-positive rate of **40–100%**, *all* of OGHARN's **41** ***newly-found*** **bugs** are semantically-valid per their APIs' intended usage. Moreover, our approach makes effective fuzzing possible where prior approaches fall short—with **15 bugs** found in **four** libraries that, until now, lacked any fuzzing harnesses.

Through the following contributions, we enable correct and thorough generation of library fuzzing harnesses, allowing state-of-the-art bug-finding to be more broadly deployed across today's countless security-critical software APIs:

- We introduce *Oracle-guided Harnessing*: a technique for fully-automatic, semantically-correct synthesis of library fuzzing harnesses. With only the minimal information found in library headers, Oracle-guided Harnessing leverages fuzzing-style mutation to construct—and refine—candidate harnesses, ensuring they uphold APIs' intended function arguments, data dependencies, and call sequences.
- We implement Oracle-guided Harnessing as a prototype implementation, OGHARN, and evaluate it on **20 diverse, real-world libraries** alongside the leading automated harness generation approach, Hopper, as well as a suite of 33 developer-written OSS-Fuzz fuzzing harnesses.
- We show Oracle-guided Harnessing upholds correctness: across our 20 real-world library benchmarks, OGHARN's **1,452** generated harnesses achieve a **0%** false-positive rate, compared to Hopper's **40–100%** false-positive bug rate.
- We demonstrate Oracle-guided Harnessing's effectiveness: across five 24-hour fuzzing campaigns, OGHARN-generated harnesses outperform developer-written harnesses by a median **14%** code coverage. Moreover, while Hopper and developers' harnesses both find **10** and **11** new security vulnerabilities, respectively, **OGHARN reveals 41**—with **40 of these since fixed** by their respective API maintainers.
- To enable future research in library harnessing, we release our prototype implementation, OGHARN, and all of our evaluation benchmarks and artifacts at the following URL: `https://github.com/FuturesLab/OGHarn`

II. BACKGROUND, RELATED WORK, AND MOTIVATION

This section provides an overview of the topics fundamental to *Oracle-guided Harnessing*: library APIs, coverage-guided fuzzing, and automated harness generation.

A. Software Library APIs

Application Programming Interfaces (APIs) make up the fundamental building blocks of today's applications and systems.[1] Often maintained as special-purpose code for distinct computing tasks (e.g., decoding a specific file format [17]), software libraries rank among the largest shares of the modern software ecosystem—with an estimated 69% of developers using third-party APIs in their own code [4]. Today, hundreds of thousands of libraries exist for a multitude of domains like multimedia [3], document processing [15], scientific computing [84], [45], and handling niche data formats [17], [90].

[1]We hereafter refer to *libraries* and *APIs* interchangeably.

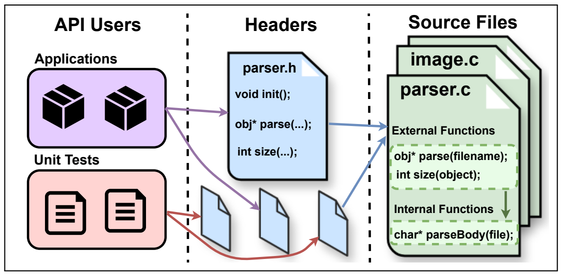
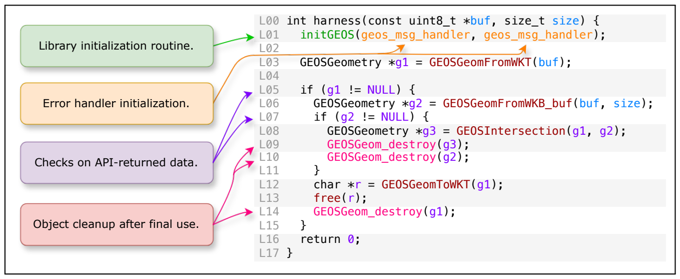


Fig. 1: High-level visualization of a typical software library's components.

Figure 1 offers a high-level overview of a typical C-based API's core components. To use the API, developers must first import its *header* files, which contain declarations—namely, argument and return types—of the library's *interface* functions: the entrypoints to invoking the library on user-provided data. These interfaces are themselves abstractions of the library's *internal* subroutines, like file I/O, data processing, and memory allocation. Though Liu et al. [60] show the feasibility of fuzzing libraries' internal functions, this broadly remains outside the scope of current large-scale library fuzzing initiatives [69] which concentrate on libraries' interfaces.

B. Fuzzing Library APIs via Harnesses

With their ever-increasing prevalence, software libraries remain a high-value target for attackers. Modern OS kernels, web browsers, and other security-critical software are a veritable spaghetti of API dependencies, creating significant risk for trickle-down vulnerabilities stemming from insecure third-party code [65]. Moreover, high-profile attempts to backdoor commodity APIs further underscores their risk to today's software ecosystem [10], [9]. To this end, library developers and maintainers have overwhelmingly turned to a proactive vulnerability-finding strategy known as *fuzzing*.


Library initialization routine.
Error handler initialization.
Checks on API-returned data.
Object cleanup after final use.

```
int harness(const uint8_t *buf, size_t size) {
  initGEOS(geos_msg_handler, geos_msg_handler);

  GEOSGeometry *g1 = GEOSGeomFromWKT(buf);

  if (g1 != NULL) {
    GEOSGeometry *g2 = GEOSGeomFromWKB_buf(buf, size);
    if (g2 != NULL) {
      GEOSGeometry *g3 = GEOSIntersection(g1, g2);
      GEOSGeom_destroy(g3);
      GEOSGeom_destroy(g2);
    }
    char *r = GEOSGeomToWKT(g1);
    free(r);
    GEOSGeom_destroy(g1);
  }
  return 0;
}
```

Fig. 2: A visualization of an expert-written library fuzzing harness (right) and its corresponding API usage-governing library semantics (left).

Having found thousands of defects across the software ecosystem, fuzzing remains by far today's most successful testing approach for bug and vulnerability discovery. Modern fuzzers (e.g., AFL++ [26]) test software by feeding it massive amounts of random inputs and measuring the code coverage (e.g., basic blocks executed) of each. By saving and mutating only inputs that reveal *new* coverage, fuzzers gradually explore more and more of the software under test, thereby revealing bugs and vulnerabilities hidden throughout its components.

Fuzzing libraries requires *harnesses*: "wrapper" programs that feed fuzzer-provided data to APIs' functions. Figure 2 shows an example harness for `libGEOS` [45]. Akin to third-party API-using applications, harnesses form the backbone of library fuzzing. Google's OSS-Fuzz [69]—the largest API fuzzing initiative—continuously fuzzes harnesses for over 1,000 APIs. Popular library fuzzers include AFL++ [26], HonggFuzz [76] and the now-retired libFuzzer [5].

Yet, **writing harnesses by hand is no easy task**. Correctly calling an API's functions demands intimate knowledge of its *semantics*: the developer-created "rules" governing its intended usage (e.g., Figure 2). For example, many APIs expose initialization routines (e.g., `initGEOS()` in Figure 2) which, if skipped, cause subsequent operations to *crash* from accessing uninitialized memory. While such crashes are possible, they are seldom realistic—triggered solely by API *misuse*—leaving them overwhelmingly unlikely in real-world software that uses the same library. For this reason, semantically-invalid crashes are virtually always dismissed by developers as false-positive bugs (e.g., [7], [8], [6]). Moreover, many types of API misuse—*missing handlers*, *unchecked data*, *premature cleanup*, and *invalid arguments*—lead many inputs to immediately fail or error-out, leaving fuzzers unable to cover more than a small fraction of the target library. Thus, manual harnessing remains unscalable, as often only a library's maintainers are best-equipped to write correct, thorough harnesses.

C. Automated Library Harnessing

To fuzz libraries in the absence of expert harnesses, many techniques have emerged for synthesizing harnesses *automatically* [12], [53], [54], [91], [20], [48]. Shown in Table I, these broadly fall under two high-level categories: (1) *static*-analysis-based harnessing; and (2) *dynamic*, on-the-fly harness generation. We weigh their trade-offs below.

Name	Harnessing Approach	Reference Agnostic	Specification Agnostic
Fudge [12]	static	✗	✓
FuzzGen [53]	static	✗	✓
Utopia [54]	static	✗	✓
Rubick [91]	static	✗	✓
GraphFuzz [48]	dynamic	✓	✗
Hopper [20]	dynamic	✓	✓

TABLE I: Current automated API harnessers by style (static or dynamic), and reliance on pre-written reference code and/or human-defined specifications.

Static Harnessers: Most harnessers today leverage static analysis, extracting API semantics from various sources of reference code before transforming them into concrete harnesses. FUDGE [12] analyzes third-party API usage among different libraries; FuzzGen [53] and Rubick [91] instead focus on API-using applications bundled *with* libraries (e.g., the `geosop` [46] program provided by `libGEOS`); and Utopia [54] searches for API usage in libraries' unit tests. Yet, while static harnessing works well for some libraries, it is seldom usable in large-scale harnessing. For many APIs, available reference code only covers a minimal set of functions, **limiting the overall coverage** of any derived harnesses. Worse yet, such approaches cannot be used on libraries that *lack* reference applications or unit tests. Moreover, these techniques are not immune to typical static analysis roadblocks [59] (e.g., indirect branches); and as recent work shows, they thus **cannot harness the vast majority of APIs** in practice [20]. To this end, recent work [48], [20] instead proposes on-the-fly harnessing by leveraging *dynamic* program analysis.

Dynamic Harnessers: Unlike static reference-code-based harnessing, dynamic harnessers instead leverage API *run-time* feedback. At a high level, they mutationally stitch-together candidate API calls and data flow, saving only those deemed valid following their execution. GraphFuzz [48] performs harnessing *during* fuzzing, using fuzzers' existing code coverage feedback to guide its mutation. Yet, while GraphFuzz eliminates need for reference code, it requires pre-defined library "schema" specifications for virtually all API semantics (e.g., types, argument ranges). Unfortunately, as GraphFuzz cannot synthesize these schemas itself—forcing users to instead write them by hand—it suffers the **same burden of human expertise** as manual harnessing.

D. Motivation: Challenges of Reference- and Specification-free Dynamic Harness Generation

To overcome GraphFuzz's limitations, Chen et al. introduce Hopper [20]: a reference- *and* specification-agnostic dynamic harnesser. Hopper also harnesses functions *during* fuzzing, adding additional heuristics to infer semantics like argument constraints, call sequences, and other usage rules typically found in user-defined specifications. Yet, while Hopper extends fuzzing's reach far beyond specification-restricted GraphFuzz, it *misses* many critical API semantics—resulting in a self-reported **false-positive crash rate upwards of 64%** [20].

To understand Hopper's limitations in dynamic harnessing, we examine its performance on `libGEOS` [45]: a popular and large computational geometry library. Our analysis of Hopper-generated harnesses reveals three high-level shortcomings spanning libraries' (1) *initialization*, (2) *data and arguments*, and (3) *call sequences*. We discuss these in detail below.

```
1  void handler(const char *fmt, ...){
2      exit(EXIT_FAILURE);
3  }
4  int main() {
5      ContextHandle_HS *c = GEOS_init_r();
6      Context_setErrorHandler_r(c, handler);
7      OrientPolygons_r(c, ...);   // Crash!
8  }
```

Fig. 3: Crash due to Hopper's failure to initialize error handler (L1–L3, L6).

Errors in Initialization: Besides calling explicit library setup routines (e.g., `initGEOS` [47]), valid library initialization must also uphold expected *error-handling* semantics. For example, in `libGEOS`, many errors—both logical and memory corruptions—are correctly caught by its under-the-hood error checking, and intentionally exposed to user-definable handler functions [47] to facilitate graceful exiting.

Yet, as Figure 3 shows, many Hopper-found crashes are only triggered due to it *missing* defining (L1–L3) and initializing (L6) handler functions. Upon incorporating the expected error handling, we see these crashes disappear. Thus, preventing false-positive crashes demands a harnessing approach that ***upholds* libraries' required initialization semantics**.

```
1   ContextHandle_HS *c = GEOS_init_r();
2   Geom_t *x = Geom_createEmptyPoint_r(c);
3   Geom_t *y = Geom_createCollection_r(c, -9, &x, 6);
4   if (y == NULL) return 0;
5   Normalize_r(c, x);    // Crash!
```

Fig. 4: Crash due to Hopper's omission of the non-`NULL` check on L4.

Errors in Data and Arguments: Libraries often require that returned objects' validity be checked before continuing execution. For example, `libGEOS` [47] states that the `createCollection` function family returns a newly-allocated geometry—and on exception, `NULL`. Correctly exiting on `NULL` is thus key to preventing crashes from future accesses of now-corrupted objects. Yet, as Figure 4 shows, Hopper *misses* such checks (L4), leading to false-positive crashes that are otherwise avoidable by upholding the API's intended data-checking semantics. Avoiding data- and argument-related false positives thus requires that **expected validity checks be *included*** for all API-returned objects.

```
1   ContextHandle_HS *c = GEOS_init_r();
2   WKBReader_t *r = WKBReader_create_r(c);
3   WKBReader_destroy(r);
4   WKBReader_destroy_r(c, r);   // Crash!
```

Fig. 5: Crash from Hopper's erroneous deallocation of deallocated data (L4).

Errors in Call Sequences: Beyond initialization and data checking, successful harnessing must also preserve libraries' intended *call sequences*. Often, inter-function dependencies are seldom explicit outside of APIs' documentation, making inferring them without reference code or specifications a compelling challenge. For example, Figure 5 highlights Hopper's inability to recognize a key restriction of `libGEOS`'s cleanup routines—namely, that already-deallocated objects cannot be targeted by *other* cleanup functions—triggering false-positive double-free crashes on its second cleanup call (L4). We further observe similar use-after-frees caused by *premature* cleanup of objects passed as arguments to other functions. Thus, evading these and other false positives demands that API harnessing **pinpoint and uphold critical *inter-function* semantics**.

Impetus: Effective library fuzzing requires harnesses that uphold APIs' key semantics—initialization, valid data and function arguments, and intended call sequences. Unfortunately, current harnessers **struggle scaling to today's large and complex libraries:** static-analysis-based approaches [53], [54] cannot recover API semantics without detailed reference code like unit tests or full-fledged applications; while dynamic ones similarly rely on expert-written reference specifications [48]—or worse—are derailed by false-positive crashes in their absence [20]. Without correct, semantics-*aware* automated harnessing, software APIs will remain indefinitely difficult to fuzz—**leaving a countless number of their vulnerabilities ripe for exploitation**.

III. Oracle-guided Automated Harnessing

Though Hopper's reference- and specification-agnostic harnessing extends fuzzing's reach far beyond prior restrictive strategies, its frequent API misuse—spanning library initialization, data, and call ordering—leaves it facing **over a 50% false-positive crash rate [20]**. Correct and thorough API fuzzing thus demands a harnessing approach that upholds API semantics—without relying on seldom-available reference code or painstakingly-written specifications.

This section introduces *Oracle-guided Harnessing*: our approach for automatically generating semantics-preserving fuzzing harnesses for library APIs. At its core, Oracle-guided Harnessing combines the trial-and-error process by which harnesses are written by hand—with fuzzing. Beginning with only the minimal information exposed in library headers, harness candidates are mutationally stitched-together and evaluated via a set of *Correctness Oracles*: compilation, execution, and coverage. By closely monitoring these side-effects, Oracle-guided Harnessing's trial-and-error mutation culls invalid, semantics-breaking mutations—converging on a final corpora of valid harnesses in as little as one hour for many real-world libraries. In the following sections, we detail Oracle-guided Harnessing's three fundamental phases (Figure 6): (1) *Pre-Processing*, (2) *Synthesis*, and (3) *Post-Processing*.

A. Phase 1: Library Header Pre-Processing

To extend harnessing to as many APIs as possible, Oracle-guided Harnessing aims to be *reference-* and *specification-*agnostic (Table I). Key to this is its *Pre-Processing* phase, which parses library headers to pinpoint functions, types, and other API-relevant objects. We discuss these steps below.

Header File Parsing: By consuming only library headers, Oracle-guided Harnessing evades the need for error-prone static code analysis or cumbersome human-written specifications. Using off-the-shelf C parsing APIs, we analyze all available headers to recover key fields such as the *types*, *names*, and *values* of header-defined macros, enums, typedefs, functions, and function pointers (Table II).

Category	Fields	Representative Example
Macros	Name	`#define TYPE_BITS 7`

Enums	Type
	Values
		`enum GeomTypes`
		`{GEOS_POINT, GEOS_LINESTRING}`

Typedefs	Type Alias	`typedef Geometry* Geom;`

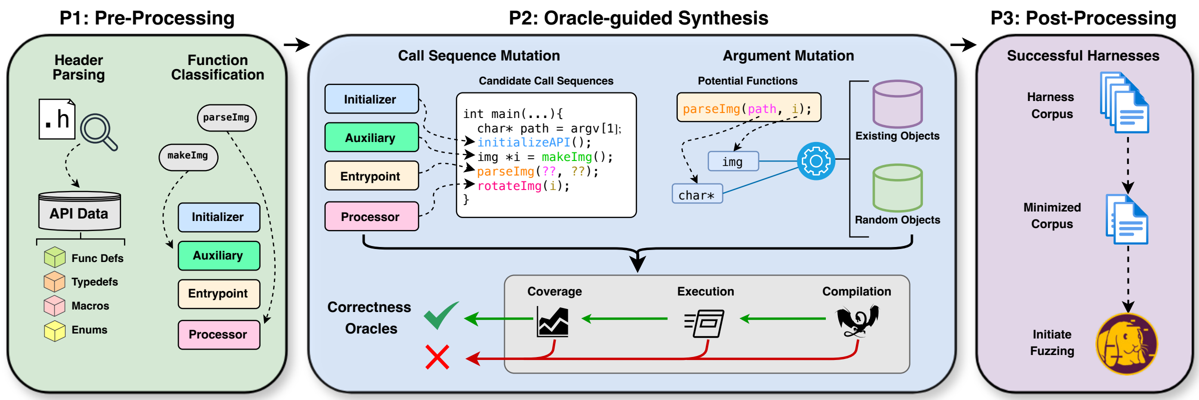
Functions	RetType
	Name
	ArgTypes
	Keywords
		`int CoordSeq_getSize`
		`(const CoordSeq* s, unsigned int* size);`

Function Pointers	RetType
	Alias
	ArgTypes
	Keywords
		`typedef void (*MessageHandler)`
		`(const char* fmt, ...);`

TABLE II: Relevant artifacts recovered from API headers.

Function Classification: With recovered functions in-hand, we perform a heuristic classification spanning four key types:

- **Initializers.** As many APIs require initialization routines, we flag functions whose names contain substring "`init`" as likely library *initializers*. For example, as Table III shows,



```
int main(...){
  char* path = argv[1];
  initializeAPI();
  img *i = makeImg();
  parseImg(??, ??);
  rotateImg(i);
}
```

```
parseImg(path, i);
```

Fig. 6: Oracle-guided Harnessing's key steps: Pre-Processing, Synthesis, and Post-Processing.

our heuristic catches `libGEOS` initializers `initGEOS` and `GEOS_init_r`. While APIs may use differently-named initializers (e.g., "`setup`"), our heuristic is easily extended to support them; however, our survey of OSS-Fuzz's **281** C APIs (§ VI) shows the vast majority of those requiring initializers—**92.2%**—maintain "`init`" naming patterns.

Function Type	Representative Examples
Initializers	`void initGEOS(...);` `ContextHandle_t GEOS_init_r(...);`
Data Entrypoints	`Geometry* *WKTReader_read (WKTReader* reader, const char* wkt);` `Geometry* GeoJSONReader_read (GeoJSONReader* reader, const char* geojson);`
Data Processors	`Geometry* Intersection (const Geometry* g1, const Geometry* g2);` `Geometry* SingleSidedBuffer (const Geometry* g, double width, int qSegs, int jStyle, double mLim, int lSide);`
Auxiliary	`WKTReader* WKTReader_create(void);` `GeoJSONReader* GeoJSONReader_create(void);`

TABLE III: Example Pre-Processing function classification.

- **Data Entrypoints.** Following initializers, we scan for possible data *entrypoints*: functions that transform fuzzer-populated buffers (e.g., filepaths or streamed data)—the common-case data source among **93.2%** of today's OSS-Fuzz C libraries (§ VI)—into the final *API-specific* objects used by the library. Table III shows two such entrypoints: `WKTReader_read` and `GeoJSONReader_read`, which consume buffers (`wkt` and `geojson`, respectively) and return pointers to new `libGEOS Geometry` objects. At a high level, different entrypoints represent the unique ways an API accepts user-provided data—laying the foundation for *how* and *where* harnesses must inject fuzzers' inputs.
- **Data Processors.** While minimal harnesses only target data entrypoints (e.g., `HDF5`'s [83]), thorough fuzzing requires extended harnesses probing an API's other data consumers (e.g., `libICAL`'s [18]). Thus, after entrypoint identification, we further scan for data *processors*: functions that perform operations on the API-specific objects returned by entrypoints. For example, in Table III, `Intersection` and `SingleSidedBuffer` are marked as processors due to their consuming of entrypoint-populated `Geometry` objects. By including data processors, our harnessing discovers a wider range of input-dependent behavior—and hard-to-find vulnerabilities—than entrypoint-only harnessing alone.
- **Auxiliary Functions.** Often, API functions expect specific arguments—I/O contexts, error/message handlers, or to-be-populated data containers—that must be initialized via special routines provided by the library, which we refer to as *auxiliary* functions. Unlike entrypoints and processors, we enforce that auxiliary functions consume *no* API-specific objects or buffered data; in other words, their sole purpose is to resolve *other* functions' arguments rather than be fuzzed themselves. For example, in Table III, auxiliary functions `WKTReader_create` and `GeoJSONReader_create` instantiate the `WKTReader` and `GeoJSONReader` handles required by aforementioned entrypoints `WKTReader_read` and `GeoJSONReader_read`, respectively.

Our library function classification aims to uphold the high-level structure of real-world fuzzing harnesses [69]: (1) library invocation, (2) data injection, and (3) data transformation. By explicitly modeling these key semantics—via initializer, entrypoint, processor, and auxiliary functions—Oracle-guided Harnessing *avoids* common harnessing pitfalls that plague prior harnessers with crash-triggering API misuse (§ II-D).

B. Phase 2: Oracle-guided Synthesis

Following header and function analysis, Oracle-guided Harnessing is ready to begin synthesizing functional harnesses. Critical to successful harnessing is pruning invalid harnesses that, if fuzzed, will trigger false-positive crashes (§ II-D). To validate harness mutations, we leverage a set of three *Correctness Oracles*: compilation, execution, and code coverage; and like a typical fuzzer [26], [68], [76], we save—and further mutate—*only* those deemed valid by our oracles. In the following, we discuss our key harness *call sequence* and *argument* mutators; and how Correctness Oracles facilitate Oracle-guided Harnessing's synthesis of thorough, valid harnesses.

Mutating Call Sequences: With function classification from Pre-Processing (§ III-A), our harness mutation begins constructing harnesses *from scratch*—beginning with initializers, then entrypoints, and lastly data processors. Our call sequence manipulation employs the following mutators:

1) **Initializer Insertion.** Should any initialization routines be identified in the library, we insert at most one initializer function at the beginning of the harness.
2) **Selecting a Data Entrypoint.** We select a preliminary entrypoint by identifying functions consuming string-like arguments (i.e., buffers, character pointers). Should none be found, we relax this constraint and further consider functions that take-in `void` pointers, as they are commonly used in real-world harnessed libraries.
3) **Fuzzer-to-Harness Input Loading.** After entrypoint selection, we incorporate necessary handling for *how* data should be loaded into derived harnesses. For filepath-consuming entrypoints, we adjust its relevant argument to `argv[1]`, where the path to fuzzer-generated inputs will be swapped-in at runtime. For stream-consuming entrypoints, we inject standard file I/O code (e.g., `fread()`) to populate its argument buffer from fuzzer-emitted files; and, if applicable, we resolve the corresponding `size` argument. In our prototype OGHARN (§ IV), we relegate it to the user to select their desired input-loading approach per library.
4) **Calling Data Processors.** Following data entrypoints, we begin inserting processor functions that consume entrypoint-returned data objects (e.g., the `Geometry` pointers in Table III). After a processor function is added, we restart mutation to attempt integrating additional processors toward fuzzing even more library functionality. While users may configure Oracle-guided Harnessing to generate increasingly-large harnesses, we observe that 2–4 processor functions per harness offers a balanced trade-off between generation time and coverage depth.
5) **Final Status Check Insertion.** Specifications often proscribe *status checks* on API-returned data: for example, that a return code is non-negative (e.g., for `GEOSNormalize`) or a returned pointer is non-`NULL` (e.g., Figure 4). As omitting such logic frequently causes spurious false-positive crashes (§ II-D), we add the following status checks to validate all API-returned data: for booleans, that they are `TRUE`; for pointers, that they are non-`NULL`; and for numerical values, that they exceed the library's common error-indicating value (e.g., `-1`, `0`). Should any checks fail, we gracefully exit the harness, ensuring that only *correct* data proceeds through to later API calls within the harness.

Mutating Arguments: Following every candidate function call insertion, Oracle-guided Harnessing initiates *argument* mutation aimed at resolving harnesses' data dependencies. We perform data resolution by leveraging all collected API artifacts alongside the following argument-level mutators:

- **Function Pointers.** For resolving function pointers like error handlers (e.g., `Context_setErrorHandler` in Figure 3), we introduce dummy stubs that merely exit once called (e.g., the `handler` function in Figure 3).
- **Data Objects.** We attempt injecting existing API-returned objects (e.g., `Geometry` from Table III); newly-initialized ones (e.g., from an auxiliary function or newly-declared struct); or `NULL`. Should *no* compatible auxiliary functions exist, we re-perform call sequence mutation to search for possible processors that return the expected type.
- **String-like Objects.** Our mutation iterates through plausible filenames for reading/writing (e.g., `/tmp/ABC123`); plausible I/O permissions (e.g., `R`, `W`); or if the argument is a plausible data *destination* (e.g., a buffer that is non-`const`), a newly-initialized object of the same type.
- **Numerical Values.** We insert known magic numbers (e.g., `0`, `1`, `-1`, `64`); macro- or `enum`-defined values (e.g., `TYPE_BITS` and `GEOS_POINT` from Table II); or `sizeof()` for buffer-adjacent integer arguments.

Validating Mutations via Correctness Oracles: Manual harness development is seldom a one-shot task. Often, successful harnessing requires iterative refinement through trial-and-error testing [13]. Typically, semantically-incorrect harnesses manifest themselves by (1) being un-compilable; or, when fuzzed, soon (2) crashing or (3) reaching premature coverage plateaus [43]. While some prior approaches validate their full-fledged harnesses *post*-generation [53], [55], [92], we see these three critical signals—compilation, execution, and coverage—as an opportunity to **automatically validate *every* incremental mutation *during* the harness synthesis process**.

To validate mutation correctness, we introduce the concept of *Correctness Oracles* (Figure 6). Given a small, pre-selected set of seed inputs for the target library on-par with those used in conventional fuzzing campaigns [52] (i.e., spanning both *correct* and *malformed* files), we evaluate the validity of each call sequence and argument mutation by performing the following three oracle-guided validity tests:

1) **Compilation.** We attempt to compile every candidate harness mutation and subsequently discard those that fail from compiler-reported errors. For many libraries, compilation failures catch the overwhelming majority of incorrect harness mutations containing semantic errors like undefined behavior, type errors, or language-level misuse.
2) **Execution.** Should compilation succeed, we attempt executing each candidate on the pre-selected corpus of correct and malformed library inputs. To ensure subtle, ordinarily-non-crashing API misuses are caught, we further leverage off-the-shelf sanitizers (e.g., AddressSanitizer [67], UndefinedBehaviorSanitizer [81]). Should *any* input crash, we discard it as a probable semantically-invalid harness. For *correct* inputs—that should otherwise pass-through harnesses error-free—we further enforce that `STDERR` remain clear, as libraries' built-in error-handling often detects and reports API misuse (e.g., Figure 3). As with manual harnessing, crashing on a limited set of test cases likely indicates API misuse, enabling Oracle-guided Harnessing to quickly filter-out many broken harness mutations.
3) **Code Coverage.** Lastly, we profile candidates' runtime code coverage to detect likely plateaus stemming from *non-crashing* API misuse. Our analysis compares coverage on (1) correct-vs-malformed inputs and (2) correct-vs-correct ones. Intuitively, APIs perform many validity checks to throw-out inputs that are malformed; and leveraging this, we enforce that correct inputs yield *higher* overall coverage of the library than malformed ones. Moreover, coverage dis-

crepancies between multiple well-formed inputs typically indicate interesting, input-*dependent* behavior; and thus, we further mandate that coverage *changes* between at least one correct input relative to its peers. As time goes on and more candidates are culled, Oracle-guided Harnessing gradually converges on valid, high-coverage harnesses.

Learning Call and Data Dependencies: When mutations are discarded for failing one or more oracles, Oracle-guided Harnessing revisits mutation to find better call and/or argument candidates. By keeping only those mutations that pass *all* Correctness Oracles, our approach eventually learns critical API semantics related to initialization, data and arguments, and call sequences (§ II-D)—resulting in **zero false positives** across our evaluation experiments (§ V).

Though Oracle-guided Harnessing needs some manual effort to find the initial correct and malformed seed inputs for the target API, we observe this is usually trivial, as most libraries make such samples publicly available in their code repositories. Nonetheless, Oracle-guided Harnessing succeeds even with few seeds: across our evaluation experiments, our per-API seed corpora range between **just 5–11 files in size**.

C. Phase 3: Final Post-Processing and Fuzzing

With Synthesis complete, Oracle-guided Harnessing's final harness corpora is minimized to reduce fuzzing resource burden. We mimic the minimization process of most fuzzers [26], [68], [76]: collecting the corpus' global coverage, and keeping the fewest samples necessary to cover its entirety. Specifically, we use conventional coverage tracing tools [26] to replay all generated harnesses on the same per-library seed files used in Correctness Oracle validation. We then calculate each harness's unique coverage, and discard those whose coverage are strict subsets of others'. Following minimization, the final reduced corpus of harnesses is ready for fuzzing.

IV. Implementation: OGHarn

We integrate Oracle-guided Harnessing as a prototype, OGHarn, spanning 2.27K lines of Python modules for parsing, mutation, and validation. We detail these below.

Parsing Implementation: We parse library headers using the Python `CXXHeaderParser` library [70], returning a final `JSON` dump containing all relevant functions, objects, and other artifacts (e.g., Table II). We further implement various Python classes to map these individual artifacts to their respective functions, arguments, and call sequences that are mutated throughout harness synthesis.

Mutation Implementation: Our harnessing operates on an internal graph-based intermediate representation (IR), facilitating easy integration of new function or argument mutators. At present, we cap mutations per each harnessed API function to 128, though this is adjustable if desired.

Validation Implementation: Following mutation, we perform validation via Correctness Oracles. OGHarn uses Python library `cfile` [49] to translate harness IR into C code. We compile harnesses with the state-of-the-art fuzzer AFL++'s [26] own compiler, injecting both sanitizers (i.e., ASAN [67] and UBSAN [81]) and coverage-tracing instrumentation into the APIs and harnesses. For compilation and execution oracles, we monitor harnesses' process signals and outputs (e.g., `STDERR`). For code coverage, we trace and process coverage using AFL++'s `afl-showmap` utility.

Supported Libraries: While our prototype targets C-based APIs, we expect future enhancements will extend Oracle-guided Harnessing to C++ and other languages. We discuss the technical trade-offs of supporting new languages in § VI.

V. Evaluation

Our evaluation of Oracle-guided Harnessing is guided by the following high-level research questions:

Q1: Are Oracle-guided Harnessing's Correctness Oracles effective at preventing API misuse?

Q2: Do Oracle-guided Harnessing's generated harnesses cover considerable library functionality?

Q3: How does Oracle-guided Harnessing enhance library fuzzing's vulnerability discovery?

A. Experiment Setup

To evaluate Oracle-guided Harnessing's strengths at dynamic harness synthesis, we compare our prototype, OGHarn, to the current state-of-the-art dynamic harnessing tool Hopper [20]. We further compare Oracle-guided Harnessing to the available developer-written OSS-Fuzz [69] harnesses per library, selecting those targeting the same header files as OGHarn (e.g., `geos_c.h` for `libGEOS`).

Following the standard set by Klees et. al [56], we run all fuzzing campaigns for five trials per competing harnessing approach. We use the state-of-the-art fuzzer AFL++ [26] as libFuzzer is no longer maintained since 2022 [5]; and accordingly modify all OGHarn and developer-written harnesses to accept AFL++ file-based input. We seed OGHarn's synthesis with 5–11 publicly-sourced files per API; and reuse these corpora as seeds for all harness fuzzing campaigns. We perform all tests and data analysis across three 24-core Ubuntu 22.04 workstations, each with 64G RAM and an Intel i9-12900K CPU. All means are calculated as geometric means.

Benchmark Selection: To ensure rigorous evaluation of OGHarn, we follow the standard of prior work and select **20 popular and actively-maintained *real-world* libraries** of varying size and type (e.g., mathematical, web, and multimedia software), shown in Table IV. For each library, **we deploy its available developer-written OSS-Fuzz [69] harnesses** per the header files considered. To assess how Oracle-guided Harnessing extends fuzzing's reach to yet-unharnessed libraries, we incorporate four benchmarks ***without* existing harnesses**: `libFYAML`, `RayLib`, `Faup`, and `StormLib`.

B. Q1: Semantic Correctness

As § III-B details, excessive crashes or premature coverage plateaus are key signs that a harness is semantically invalid

Library	Header File(s)	# Funcs	Manual (OSS-Fuzz) Harnesses
libGEOS [45]	geos_c.h	525	fuzz_geo2 [44]
HDF5 [84]	hdf5.h	114	extended_fuzzer [82] read_fuzzer [83]
libICAL [17]	ical.h	256	fuzzer [19] extended_fuzzer [18]
OpenEXR [30]	ImfCRgbaFile.h	86	exrcheck_fuzzer [28] exrcorecheck_fuzzer [29]
Lexbor [15]	html.h	331	document_parse [16]
libUCL [75]	ucl.h	123	add_string_fuzzer [73] msgpack_fuzzer [74]
cglTF [58]	cgltf.h	38	fuzz_main [57]
GPAC [3]	isomedia_dev.h constants.h	343	fuzz_parse [2]
C-Ares [78]	ares.h	77	parse_reply_fuzzer [80] create_query_fuzzer [79]
cJSON [42]	cJSON.h	78	cjson_read_fuzzer [41]
LCMS [64]	lcms2.h	290	it8_load_fuzzer [61] overwrite_transform_fuzzer [62] transform_fuzzer [63]
libMagic [22]	file.h magic.h	93	magic_fuzzer [23]
libPCAP [85]	pcap.h	79	fuzz_both [86] fuzz_filter [87] fuzz_pcap [88]
PCRE2 [51]	pcre2.h	120	pcre2_fuzzsupport [50]
SQLite [71]	sqlite3.h	287	oss_fuzz [72]
Zlib [31]	zlib.h	81	checksum_fuzzer [32] compress_fuzzer [33] example_dict_fuzzer [34] example_flush_fuzzer [35] example_large_fuzzer [36] example_small_fuzzer [37] minigzip_fuzzer [38] uncompress_fuzzer [40] uncompress2_fuzzer [39]
libFYAML [11]	libfyaml.h	387	None available.
RayLib [77]	raylib.h	549	None available.
Faup [1]	decode.h features.h faup.h options.h	45	None available.
StormLib [90]	StormLib.h	76	None available.

TABLE IV: Our 20 real-world library benchmarks; their targeted header files and the total functions they contain; and available expert-written harnesses.

Library	# OGHARN Harnesses	# Harnesses with Symptoms of API Misuse: Frequent Crashes	Coverage Plateau	Final
libGEOS	120	1	3	0
HDF5	57	1	3	0
libICAL	62	0	0	0
OpenEXR	9	0	0	0
Lexbor	94	0	1	0
libUCL	67	0	0	0
cglTF	34	0	0	0
GPAC	120	0	0	0
C-Ares	56	0	0	0
cJSON	112	0	0	0
LCMS	65	0	0	0
libMagic	18	0	0	0
libPCAP	45	0	0	0
PCRE2	45	0	8	0
SQLite	119	0	21	0
Zlib	53	0	0	0
libFYAML	118	0	0	0
RayLib	120	0	0	0
Faup	18	0	0	0
StormLib	120	0	0	0
			OGHARN's Confirmed API Misuses:	**0**

TABLE V: OGHARN's signs of possible API misuse seen in 1-hour fuzzing trials. We investigate all by hand and confirm all of these harnesses indeed uphold *valid* API semantics. We cap OGHARN's synthesis at 120 harnesses per API; for smaller APIs, our synthesis results in fewer overall harnesses.

and *misusing* its targeted API. To examine whether Oracle-guided Harnessing is successful generating valid harnesses, we perform a smoke-test evaluation on up to 120 OGHARN-generated harnesses per library and fuzz them each for 5×1-hour trials.[2] We flag any harnesses where at least 10% of all inputs crash, or where coverage does not increase in either the first or second half of the campaign. Referencing APIs' documentation, we then manually analyze all flagged harnesses to make a final diagnosis as to whether API misuse occurs.

Table V lists our observed potential misuses stemming from abnormal crash rates or coverage. We see **38** suspicious instances out of the **1,452 total harnesses** generated across all 20 APIs. Our manual assessment of each is as follows:

- *libGEOS*. We observe OGHARN's three code coverage plateaus in harnesses calling functions DifferencePrec, RelatePattern, and SingleSidedBuffer. In all three, OGHARN uses a single object to fill *two* matching-type arguments, which—while valid API use—limits their runtime coverage. Interestingly, OGHARN triggers a crash when supplying large buffers to function RelatePatternMatch, which expects two DE9IM-compliant (i.e., nine-length) buffers as arguments. However, this crash occurs when the *first* of these two buffers exceeds nine in size, as libGEOS ultimately performs a size check only on the *second* buffer—and misses the first. We thus conclude that OGHARN's harness is not API misuse, but rather, an exposure of a missed—yet intended—internal data check. Upon reporting, this bug was subsequently patched.[3]
- *HDF5*. As with libGEOS, three HDF5 harnesses exhibit coverage plateaus from benign data: two when calling H5Fis_accessible, and the third from H5Fis_hdf5. In manually inspecting their source code, we see that these functions all serve as basic input-checking routines that lack any substantial *input-dependent* logic, hence causing plateaus from early coverage saturation. We also catch one crash on function H5Fget_name, and find it is merely an error-handling signal from OGHARN's correctly-initialized random string being detected as an invalid file path.
- *Lexbor*. In examining one code coverage plateau revealed in Lexbor's HTML parsing function html_document_css_customs_id, we determine OGHARN's harness as valid from cross-referencing identical calls in the API's source. As with HDF5, we observe few input-dependent paths in it and its callees, indicating plateaus are likely caused by saturated coverage.
- *PCRE2*. We observe seven PCRE2 coverage plateaus due to functions with low input-dependent behavior (e.g., pcre2_get_match_data_size), and an eighth resulting from OGHARN's injection of parameters which—while valid—*constrain* string parsing: for example, though pcre2_compile parses input strings of any size, OGHARN sets a size of *one*. Ultimately, this plateauing harness is appropriately de-ranked by OGHARN's coverage-based minimization (§ III-C), with OGHARN converging on *full* string sizes in its other higher-ranked PCRE2 harnesses.
- *SQLite*. We see 17 of SQLite's 21 coverage plateaus stem from functions with low input-dependent behavior (e.g., sqlite3_changes64). Like with PCRE2, we see three others caused by arguments that, while valid, limit coverage: for example, calling sqlite3_db_filename with a filename non-existent in the SQL database, causing premature exit. Lastly, one plateau occurs

[2]Because OGHarn performs coverage-based minimization on its final harness corpora (§ III-C), some APIs attain *fewer* than 120 harnesses in total.

[3]https://github.com/libgeos/geos/issues/1084

from a valid yet unproductive call sequence, where OGHARN calls `sqlite3_db_cacheflush` right after `sqlite3_open`; as no operations occur in between to modify the database, coverage quickly stalls in the latter function. Though `SQLite` faces the most plateaus, none result from true API misuse—and as in all other APIs, OGHARN successfully converges on more interesting arguments and call sequences in its higher-ranked harnesses. Thus, as these symptoms are ultimately benign and unrelated to API misuse, we conclude that Oracle-guided Harnessing *does* produce valid harnesses. While API misuse may manifest *later* in fuzzing, our 24-hour campaigns with OGHARN's harnesses (§ V-D) reveal ***zero* false-positive crashes**—further underscoring Oracle-guided Harnessing's effectiveness.

Library	OGHARN Mutations		% Discarded Mutations by Correctness Oracle		
	# Success	# Discard	Compilation	Execution	Coverage
`libGEOS`	550	338,161	26.7%	1.7%	71.6%
`HDF5`	76	1,168,426	93.9%	5.3%	0.8%
`libICAL`	165	51,553	40.8%	54.5%	4.7%
`OpenEXR`	67	594,405	36.0%	0.9%	63.1%
`Lexbor`	143	14,596	26.1%	6.0%	67.8%
`libUCL`	108	74,589	14.1%	58.7%	27.2%
`cglTF`	39	796	17.8%	15.5%	66.7%
`GPAC`	161	796,618	77.9%	0.7%	21.3%
`C-Ares`	108	438,217	35.2%	6.0%	58.8%
`cJSON`	145	5,399	25.6%	8.5%	65.9%
`LCMS`	78	381,244	8.8%	44.2%	47.0%
`libMagic`	18	36,471	93.9%	0.4%	5.7%
`libPCAP`	65	559,103	46.1%	33.9%	20.0%
`PCRE2`	161	154,457	97.6%	0.7%	1.7%
`SQLite`	119	342,282	36.6%	9.6%	53.8%
`Zlib`	82	85,157	1.7%	6.0%	92.3%
`libFYAML`	158	338,141	97.7%	1.9%	0.3%
`RayLib`	36	1,219,870	77.6%	0.1%	22.3%
`Faup`	30	268	18.7%	47.8%	33.6%
`StormLib`	99	275,886	12.6%	9.9%	77.5%
Mean Proportion of Discards:			**31.1%**	**5.2%**	**20.3%**

TABLE VI: Statistics on OGHARN's mutations over 24 hours of synthesis.

To understand how Oracle-guided Harnessing's synthesis is influenced by our *Correctness Oracles* (§ III-B), we instrument OGHARN to log key harnessing statistics, shown in Table VI. Though we expect *execution* to catch the majority of invalid harnesses across all target libraries, we observe that oracle distribution is largely *target-dependent*: for example, as Table VI shows, execution makes up 58.7% of oracle violations for `libUCL`, while coverage represents 77.5% for `StormLib`. Thus, given OGHARN's accuracy, we conclude that Oracle-guided Harnessing's reliance on *multiple* oracles is a critical filter for invalid mutations—**preventing the API misuse** that hinders competing dynamic harnessers (e.g., § II-D).

Q1: Oracle-guided Harnessing's three oracles—compilation, execution, and coverage—are key to preventing API misuse, with ***zero* false-positive crashes** found across our entire evaluation.

C. Q2: Depth in Library Coverage

To understand whether Oracle-guided Harnessing succeeds in harnessing more API functions, we enumerate the unique calls contained in OGHARN's valid synthesized harnesses versus those in manually-written ones (Table VII). Overall, we see that OGHARN improves function coverage by a median **653% over manually-written OSS-Fuzz harnesses**.

Library	Total Functions	Manually Harnessed	OGHARN Harnessed	
`libGEOS`	525	8	**223**	27.88 ×
`HDF5`	114	6	**53**	8.83 ×
`libICAL`	256	4	**119**	29.75 ×
`OpenEXR`	86	1	**39**	39.00 ×
`Lexbor`	331	3	**154**	51.33 ×
`libUCL`	123	6	**71**	11.83 ×
`cglTF`	38	3	**14**	4.67 ×
`GPAC`	343	2	**163**	81.50 ×
`C-Ares`	77	46	**55**	1.20 ×
`cJSON`	78	6	**74**	12.33 ×
`LCMS`	290	15	**66**	4.40 ×
`libMagic`	93	3	**10**	3.33 ×
`libPCAP`	79	9	**56**	6.22 ×
`PCRE2`	120	15	**24**	1.60 ×
`SQLite`	287	17	**95**	5.59 ×
`Zlib`	81	27	**36**	1.33 ×
`libFYAML`	387	`n/a`	**119**	–
`RayLib`	549	`n/a`	**91**	–
`Faup`	45	`n/a`	**30**	–
`StormLib`	76	`n/a`	**46**	–
OGHARN's Median Relative Total Functions Harnessed:				**7.53 ×**

TABLE VII: Total API vs. manually- and OGHARN-harnessed functions (max 24-hr synthesis time). `n/a` = no available manually-written harnesses.

Yet, harnessing more functions does not guarantee higher *runtime* coverage. To validate OGHARN's runtime coverage, we fuzz OGHARN's harnesses alongside all manual harnesses and state-of-the-art dynamic harnesser, Hopper; and collect all harnesses' per-API basic block coverage by replaying their fuzzer-generated inputs via AFL-QEMU-Cov [25]. For Hopper, we use its provided translator to convert its interpreter harnesses to compilable C harnesses, with minor patching to correct syntax errors, before tracing the coverage of each. As fuzzing OGHARN's entire harnesses corpus is computationally overwhelming, we limit fuzzing to only its *top-10* highest-coverage harnesses per API. Table VIII reports OGHARN's mean relative coverage across five trials per benchmark. Following Klees et al. [56], we assess statistical significance via the Mann-Whitney U test at the $p = 0.05$ significance level.

Library	OGHARN's Coverage vs. Hopper				OGHARN's Coverage vs. Manual			
	Total	MWU p	Unique	MWU p	Total	MWU p	Unique	MWU p
`libGEOS`	1.26 ×	0.151	1.92 ×	**0.016**	2.66 ×	**0.008**	5.94 ×	**0.008**
`HDF5`	0.72 ×	**0.008**	0.28 ×	**0.008**	1.37 ×	**0.008**	19.13 ×	**0.008**
`libICAL`	0.43 ×	**0.012**	0.09 ×	**0.008**	1.55 ×	**0.012**	182.85 ×	**0.012**
`OpenEXR`	0.52 ×	**0.008**	0.04 ×	**0.008**	0.60 ×	**0.008**	0.06 ×	**0.008**
`Lexbor`	19.56 ×	**0.012**	22.16 ×	**0.008**	1.89 ×	**0.008**	28.08 ×	**0.008**
`libUCL`	0.72 ×	**0.034**	0.30 ×	0.056	1.16 ×	0.081	3.34 ×	0.095
`cglTF`	44.89 ×	**0.008**	13.64 ×	**0.009**	1.08 ×	**0.012**	73.31 ×	**0.011**
`GPAC`	`n/a`	–	`n/a`	–	1.20 ×	**0.012**	98.04 ×	**0.012**
`C-Ares`	0.63 ×	**0.011**	0.31 ×	**0.008**	0.77 ×	**0.011**	0.29 ×	**0.008**
`cJSON`	0.41 ×	**0.007**	0.01 ×	**0.012**	1.53 ×	**0.004**	6.60 ×	**0.007**
`LCMS`	0.24 ×	**0.008**	0.02 ×	**0.008**	1.08 ×	**0.008**	0.84 ×	0.151
`libMagic`	1.13 ×	0.151	0.12 ×	**0.008**	1.29 ×	**0.008**	20.29 ×	**0.008**
`libPCAP`	1.46 ×	**0.008**	0.81 ×	0.222	0.91 ×	**0.008**	0.76 ×	0.095
`PCRE2`	0.30 ×	**0.008**	0.01 ×	**0.008**	0.44 ×	**0.008**	0.03 ×	**0.008**
`SQLite`	0.64 ×	**0.008**	0.09 ×	**0.008**	1.11 ×	0.095	1.91 ×	0.421
`Zlib`	0.54 ×	**0.011**	0.01 ×	**0.012**	0.45 ×	**0.011**	0.01 ×	**0.012**
`libFYAML`	2.33 ×	**0.008**	5.06 ×	**0.008**	`n/a`	–	`n/a`	–
`RayLib`	0.69 ×	0.656	0.59 ×	0.548	`n/a`	–	`n/a`	–
`Faup`	0.91 ×	**0.025**	0.73 ×	0.205	`n/a`	–	`n/a`	–
`StormLib`	0.49 ×	**0.008**	0.12 ×	**0.008**	`n/a`	–	`n/a`	–
OGHARN's Median Relative Coverage:	**0.69 ×**		**0.28 ×**		**1.14 ×**		**4.64 ×**	

TABLE VIII: OGHARN's *total* and *unique* coverage over Hopper and manual harnesses in 5×24-hr campaigns. Statistically-significant changes (i.e., Mann-Whitney U test $p < 0.05$) are **bold**. `n/a` = API unsupported by that approach.

As Table VIII shows, OGHARN sees median total coverage **0.69**× and **1.14**× that of Hopper and developers' harnesses, respectively. To assess these approaches' coverage, we further compute and report their total *unique* basic blocks covered. Overall, OGHARN's median unique coverage is **0.28**× and **4.64**× that of Hopper and developer harnesses, respectively.

We see OGHARN generally attains higher coverage on *larger* APIs with more functions like `libGEOS` and `Lexbor` (Table VII). For smaller APIs like `C-Ares` and `Zlib` with 77 and 81 functions, respectively, OGHARN sees lower relative increases in functions harnessed—1.20× and 1.33×, respectively—suggesting these smaller APIs' OSS-Fuzz harnesses are already robust. Nonetheless, OGHARN sees 8–166% coverage improvements on **11 of 16** OSS-Fuzz APIs.

Though Hopper sees higher coverage on many benchmarks, its harnessing limitations leave it with overwhelmingly poor performance on several (e.g., `cglTF` and `Lexbor`). In analyzing these cases by hand, we find that Hopper's internal synthesis cannot properly recover key constraints in both APIs, leaving it incapable of generating any valid harnesses—**despite OGHARN having successfully harnessed them**. Moreover, we observe that Hopper is unsuccessful at harnessing `GPAC`, causing it to crash immediately from runtime errors. Thus, we conclude that Oracle-guided harnessing trades-off some level of coverage for a higher overall harnessing accuracy.

Q2: Oracle-guided Harnessing surpasses developer-written harnesses in function and runtime coverage—extending **correct *and* thorough** harnessing to APIs where other approaches fall short.

D. Q3: Vulnerability Discovery

Lastly, we assess whether Oracle-guided Harnessing's generated harnesses improve API fuzzing bug-finding. We follow our procedure from § V-C: for each library, we run 5×24-hour fuzzing campaigns per Hopper, its OSS-Fuzz harnesses, and the top-10 highest-coverage OGHARN harnesses. After fuzzing, we perform an initial deduplication of all fuzzer-found crashes using the `CASR` [66] crash triage framework on ASAN [67] and UBSAN [81] error reports; and perform additional manual deduplication and analysis of potential false-positives before **reporting all bugs to their developers**.

Library	Manual Bugs			Hopper Bugs			OGHARN Bugs		
	True	False	Unique	True	False	Unique	True	False	Unique
`libGEOS`	0	0	0	4	6	3	5	0	4
`HDF5`	5	0	0	1	2	1	7	0	2
`libICAL`	1	0	0	1	2	1	2	0	1
`OpenEXR`	0	0	0	0	0	0	0	0	0
`Lexbor`	0	0	0	0	0	0	3	0	3
`libUCL`	4	0	3	1	13	0	6	0	5
`cglTF`	0	0	0	0	0	0	0	0	0
`GPAC`	0	0	0	`n/a`	`n/a`	`n/a`	0	0	0
`C-Ares`	0	0	0	0	2	0	0	0	0
`cJSON`	0	0	0	0	2	0	0	0	0
`LCMS`	0	0	0	0	2	0	0	0	0
`libMagic`	0	0	0	0	2	0	0	0	0
`libPCAP`	0	0	0	0	1	0	0	0	0
`PCRE2`	0	0	0	0	4	0	3	0	3
`SQLite`	0	0	0	0	1	0	0	0	0
`Zlib`	1	0	1	0	4	0	0	0	0
`libFYAML`	`n/a`	`n/a`	`n/a`	0	0	0	2	0	2
`RayLib`	`n/a`	`n/a`	`n/a`	1	1	1	1	0	1
`Faup`	`n/a`	`n/a`	`n/a`	0	0	0	0	0	0
`StormLib`	`n/a`	`n/a`	`n/a`	2	12	2	12	0	11
TOTAL:	**11**	**0**	**4**	**10**	**54**	**8**	**41**	**0**	**32**

TABLE IX: OGHARN's total uncovered *true*-positive bugs, *false*-positive bugs, and *uniquely-found* true-positive bugs vs. Hopper and manually-written harnesses in 5×24-hour campaigns. `n/a` = API unsupported by that approach.

Table IX shows the true- and false-positive bugs found by all competitors. Overall, OGHARN sees the highest bug discovery, with **41 *new* true-positive bugs**—and the most unique bugs—across 9 APIs. Comparatively, developers' harnesses and Hopper reveal only 11 and 10 bugs each, respectively.

We also observe Hopper's false-positive rate between **40–100%**—within the range reported by its authors [20]—suggesting it faces significant challenges in harnessing real-world APIs. In examining Hopper's **54 false positives**, we see many of the same pitfalls as discussed in § II-D: broken initialization (**3 total**), invalid data or arguments (**45 total**), and improper call sequences (**6 total**). Developer responses on Hopper-found crashes highlight clear instances of API misuse: for example, several cases in `StormLib` involve operations on a single buffer, yet Hopper erroneously provides *different* corresponding buffer sizes to each; while others center on Hopper's injection of "garbage" values that violates their APIs' data initialization semantics. Thus, while Hopper gains higher coverage on many benchmarks (Table VIII), its semantically-incorrect harnessing frequently breaks-down on real-world libraries, leading to significant false-positive crashes in fuzzing.

Library	Error Type	Location	Manual	Hopper	OGHarn	Status
`libGEOS`	Stack BoF	`Coordinate.h:355`	✗	✗	✓	✓
	Illegal Read	`Polygon.cpp:192`	✗	✗	✓	✓
	Illegal Read	`CGAlgorithmsDD.cpp:90`	✗	✓	✓	👍
	Illegal Read	`Centroid.cpp:139`	✗	✗	✓	✓
	Heap BoF	`LineIntersector.h:604`	✗	✗	✓	✓
`HDF5`	Heap BoF	`H5HLcache.c:252`	✓	✗	✓	✓
	Large Alloc	`H5MM.c:87`	✓	✗	✓	✓
	Heap BoF	`H5MM.c:115`	✓	✗	✓	✓
	Stack BoF	`vasprintf.c:45`	✓	✗	✓	✓
	Illegal Read	`H5Oint.c:1070`	✗	✗	✓	✓
	Heap BoF	`H5Fio.c:515`	✗	✗	✓	✓
	Large Alloc	`H5Centry.c:1015`	✓	✗	✓	✓
`libICAL`	Illegal Read	`icalcomponent.c:2572`	✓	✗	✓	✓
	Illegal Read	`icalcomponent.c:2531`	✗	✗	✓	✓
`Lexbor`	Illegal Read	`element.c:370`	✗	✗	✓	✓
	Illegal Read	`avl.c:470`	✗	✗	✓	✓
	Heap BoF	`str.c:439`	✗	✗	✓	✓
`libUCL`	Illegal Read	`ucl_parser.c:77`	✗	✗	✓	✓
	Illegal Write	`ucl_parser.c:1903`	✗	✓	✓	✓
	Illegal Read	`ucl_hash.c:117`	✗	✗	✓	✓
	Heap BoF	`ucl_hash.c:460`	✓	✗	✓	✓
	Illegal Read	`ucl_util.c:3163`	✗	✗	✓	✓
	Stack BoF	`glob.c:568`	✗	✗	✓	✓
`libFYAML`	Illegal Read	`fy-list.h:198`	✗	✗	✓	✓
	Illegal Read	`fy-utf8.h:81`	✗	✗	✓	✓
`PCRE2`	Heap BoF	`pcre2_compile.c:4014`	✗	✗	✓	✓
	Heap BoF	`pcre2_compile.c:4675`	✗	✗	✓	✓
	Heap BoF	`pcre2_compile.c:4306`	✗	✗	✓	✓
`RayLib`	Illegal Read	`rtextures.c:497`	✗	✗	✓	✓
`StormLib`	Heap BoF	`SBaseFileTable.cpp:130`	✗	✗	✓	✓
	Stack BoF	`SBaseFileTable.cpp:138`	✗	✗	✓	✓
	Large Alloc	`SBaseFileTable.cpp:1137`	✗	✗	✓	✓
	Heap BoF	`SBaseFileTable.cpp:1150`	✗	✗	✓	✓
	Heap BoF	`SBaseFileTable.cpp:1799`	✗	✗	✓	✓
	Heap BoF	`SBaseFileTable.cpp:3095`	✗	✗	✓	✓
	Heap BoF	`FileStream.cpp:249`	✗	✗	✓	✓
	Divide by 0	`FileStream.cpp:1205`	✗	✗	✓	✓
	Divide by 0	`SBaseCommon.cpp:1246`	✗	✗	✓	✓
	Heap BoF	`huff.cpp:281`	✗	✗	✓	✓
	Heap BoF	`sparse.cpp:266`	✗	✗	✓	✓
	Heap BoF	`SBaseCommon.cpp:512`	✗	✗	✓	✓
					OGHARN's total fixed and unfixed-yet-confirmed vulnerabilities:	**41**

TABLE X: OGHARN's newly-found vulnerabilities in 5×24-hour campaigns. ✓ = bugs fixed after our disclosure; 👍 = bugs confirmed and awaiting fixes.

Table X lists OGHARN's 41 newly-found vulnerabilities by type, source location, and reporting status. So far, **40 of these are now fixed**, with **an additional 1 confirmed** and awaiting fixes by its developers. Overall, Oracle-guided Harnessing's semantically-valid, thorough harnesses demonstrably enhance fuzzing's reach to find many more vulnerabilities across a wider range of complex software libraries.

Q3: Oracle-guided Harnessing's synergy of iterative harness mutation and validation synthesizes correct, high-coverage library fuzzing harnesses **that greatly improve library bug discovery**.

VI. Threats to Validity

Below we weigh several limitations of Oracle-guided Harnessing and our prototype implementation, OGHarn.

Mitigation of Confirmation Bias and Overfitting: To mitigate risk of bias and overfitting, we evaluate OGHarn on 20 diverse real-world APIs spanning varied software domains, file formats, and complexity. Notably, our corpus includes **four** APIs that lack any pre-existing developer-written harnesses, allowing us to test OGHarn's robustness and adaptability in scenarios with zero prior guidance. To ensure a rigorous comparison with state-of-the-art dynamic harnessing approach Hopper, we integrate seven libraries from Hopper's original benchmark suite (`C-Ares`, `cJSON`, `LCMS`, `libMagic`, `libPCAP`, `SQLite`, and `Zlib`). This inclusion not only benchmarks OGHarn on well-documented baselines, but also highlights its performance relative to today's leading auto-harnessing approach—underscoring OGHarn's low false-positive rate and improved library bug-finding capabilities.

Generality of Function Classification: Correctly upholding APIs' semantics necessitates obeying their unique functions' distinct purposes. To this end, Oracle-guided Harnessing pre-processes library headers (§ III-A) to conservatively classify functions into four semantic types (Table III): (1) initializers, data (2) entrypoints and (3) processors, and (4) auxiliary. Our classification is based on common-case conventions seen in OSS-Fuzz's **281** C libraries [69]: of the 77 initializer-requiring APIs, 71 (**92.2%**) follow "`init`" naming patterns; while 262 of all 281 APIs (**93.2%**) support filename- or buffer-based data entry. Handling less-common conventions (e.g., file-descriptor-based data entry, non-"`init`" initializers) requires minimal extension to our prototype, which we leave to future work.

Generality of Data Status Checks: As § III-B details, Oracle-guided Harnessing avoids API misuse via aggressive *status checks*: for example, enforcing a returned pointer as non-`NULL` prevents false-positive dereference crashes in future functions accessing it. While pointers, booleans, and strings are trivial to check for validity, API-returned *numerical* values may indicate different status codes. Yet, in examining our 20 benchmark libraries' 3,978 total functions, we see that negative return values form the most common numerical value for indicating errors (i.e., 73.2% of all). Thus, while our current prototype checks only that returned numerical values are non-negative, we expect that recovering *function-specific* error values is achievable within Oracle-guided Harnessing's iterative harnessing: namely, by intentionally injecting invalid arguments and monitoring their effects on returned values. We anticipate such approach will also be effective for other cases of unexpected return semantics, such as swapped `true` and `false` roles in boolean status checks. Interestingly, we see that many validity checks are irrelevant because their functions are *inconsequential* to library state or data: for example, as `GEOSGeomTypeId` consumes only `const` data, later functions on this same data will be unaffected by trickle-down errors. We leave exploring this to future work.

Supporting Struct-based Data Injection: While Oracle-guided Harnessing covers many more functions than expert-written harnesses (Table VII), it does not harness 100% of library functions. Unlike Hopper, which supports direct population of `struct` members, OGHarn strictly populates `structs` via available API routines (e.g., `ucl_parser_new`); and consequently, it cannot harness functions that consume `structs` with *no* available initializers: for example, `cglTF` lacks any routine returning a `cgltf_node` pointer, causing functions dependent on them (e.g., `cgltf_node_index`) to go unharnessed. Similarly, libraries requiring `struct`-based *entrypoints* (e.g., `libAOM`, `libPNG`, `libVPX`) are unsupported at this time. We anticipate handling `struct` objects requires future extensions to OGHarn's data resolution (§ III-B), which already supports random initialization of common C data types—offering a means to directly populate individual `struct` members.

Supporting Server-client Libraries: Like Hopper, OGHarn also does not support most server-client libraries as they often rely on complex, multi-function setup routines. For example, `Nginx`'s OSS-Fuzz harness `http_request_fuzzer` [24] invokes *ten* distinct initializer functions. While server-client libraries are less common—only 6.05% of OSS-Fuzz's 281 C APIs according per our survey—we expect supporting them requires substantial re-engineering of OGHarn, such as: (1) enabling handling of multiple initializers as well as (2) determining their correct initializer sequence ordering. We leave exploring support for server-client libraries to future work.

Supporting Other Languages: Our preliminary C++ support targets class methods as candidate functions during harness synthesis, but needs further engineering to cover more C++ constructs such as overloaded functions.While we believe other statically-typed languages such as Java will be easier to accommodate than dynamically-typed ones, we see an opportunity to use Python's *Type Checker* as a Correctness Oracle—analogous to our use of compilation success for C harnesses. Though we expect that Oracle-guided Harnessing is amenable to these and other languages, we leave the requisite engineering and re-evaluation to future work.

VII. Conclusion

Oracle-guided Harnessing extends practical, semantics-preserving harnessing to today's many libraries that, until now, remained outside fuzzing's reach from being unharnessed. By eliminating need for reference code or user-defined specifications—without sacrificing accuracy—Oracle-guided Harnessing brings greater flexibility in harnessing toward improved API bug discovery. We demonstrate its potential on real-world APIs: revealing 41 new bugs, zero false positives, and far higher code coverage than manual harnesses alone.

Through lightweight heuristics and iterative, trial-and-error mutation, we show that harnessing can be automated with relatively high precision. We envision a new era in which traditionally-manual fuzzing tasks will become fully automated, allowing maintainers to shift efforts away from endeavouring to find bugs—and toward fixing them.

Acknowledgments

We thank our reviewers for helping us improve the paper. This material is based upon work supported by the National Science Foundation under Grant No. 2419798.

References

[1] faup: Fast URL decoder library. URL: https://stricaud.github.io/faup/.

[2] GPAC: gpac_fuzz_parse harness. URL: https://github.com/gpac/testsuite/blob/master/oss-fuzzers/fuzz_parse.c.

[3] GPAC: Ultramedia oss for video streaming & next-gen multimedia transcoding, packaging & delivery. URL: https://github.com/gpac/gpac.

[4] 19th Developer Economics Survey. Technical report, SlashData, 2020. URL: https://www.developernation.net/resources/reports/.

[5] libfuzzer Development Status, 2022. URL: https://llvm.org/docs/LibFuzzer.html\#status.

[6] libpng: false-positive crash caused by improper function call ordering, 2022. URL: https://github.com/pnggroup/libpng/issues/458.

[7] libgeos: false-positive crash caused by improper initialization, 2023. URL: https://github.com/libgeos/geos/issues/996.

[8] Sqlite3: false-positive crash caused by improper data initialization, 2023. URL: https://sqlite.org/forum/forumpost/7ace1408b.

[9] Social Engineering Takeovers of Open Source Projects. Technical report, Open JS Foundation, 2024. URL: https://openjsf.org/blog/openssf-openjs-alert-social-engineering-takeovers.

[10] XZ Utils backdoor. Technical report, The Tukaani Project, 2024. URL: https://tukaani.org/xz-backdoor/.

[11] Pantelis Antoniou. libFYAML: A fancy 1.2 YAML and JSON parser/writer. URL: https://github.com/pantoniou/libfyaml.

[12] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *ACM Joint Meeting on Foundations of Software Engineering*, FSE, 2019.

[13] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3), 2021.

[14] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *ACM Joint Meeting on Foundations of Software Engineering*, FSE, 2020.

[15] Alexander Borisov. Lexbor: Crafting a browser engine with simplicity and flexibility. URL: https://github.com/lexbor/lexbor.

[16] Alexander Borisov. Lexbor: lexbor_document_parse harness. URL: https://github.com/lexbor/lexbor/blob/master/test/fuzzers/lexbor/html/document_parse.c.

[17] Eric Busboom, Art Cancro, and Wilfried Goesgens. libICAL: an implementation of icalendar protocols and data formats. URL: https://libical.github.io/libical/.

[18] Eric Busboom, Art Cancro, and Wilfried Goesgens. libICAL: ical_extended_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/libical/libical_extended_fuzzer.cc.

[19] Eric Busboom, Art Cancro, and Wilfried Goesgens. libICAL: ical_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/libical/libical_fuzzer.cc.

[20] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. Hopper: Interpretative fuzzing for libraries. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2023.

[21] Addison Crump, Andrea Fioraldi, Dominik Maier, and Dongjia Zhang. Libafl_libfuzzer: Libfuzzer on top of libafl. In *IEEE/ACM International Workshop on Search-Based and Fuzz Testing*, SBFT, 2023.

[22] Ian Darwin. libMagic magic_fuzzer harness. URL: https://www.darwinsys.com/file/.

[23] Ian Darwin. libMagic magic_fuzzer harness. URL: https://github.com/file/file/blob/master/fuzz/magic_fuzzer.c.

[24] F5 Inc. NGINX http_request_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/nginx/fuzz/http_request_fuzzer.cc.

[25] Andrea Fioraldi. afl-qemu-cov, 2024. URL: https://github.com/andreafioraldi/afl-qemu-cov.

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*, WOOT, 2020.

[27] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2022.

[28] Academy Software Foundation. OpenEXR: exr_exrcheck_fuzzer harness. URL: https://github.com/AcademySoftwareFoundation/openexr/blob/main/src/test/OpenEXRFuzzTest/oss-fuzz/openexr_exrcheck_fuzzer.cc.

[29] Academy Software Foundation. OpenEXR: exr_exrcorecheck_fuzzer harness. URL: https://github.com/AcademySoftwareFoundation/openexr/blob/main/src/test/OpenEXRFuzzTest/oss-fuzz/openexr_exrcorecheck_fuzzer.cc.

[30] Academy Software Foundation. OpenEXR: specification and reference implementation of the EXR file format. URL: https://openexr.com/en/latest/.

[31] Jean-loup Gailly and Mark Adler. Zlib: A massively spiffy yet delicately unobtrusive compression library. URL: https://zlib.net/.

[32] Jean-loup Gailly and Mark Adler. Zlib: checksum_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/checksum_fuzzer.c.

[33] Jean-loup Gailly and Mark Adler. Zlib compress_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/compress_fuzzer.c.

[34] Jean-loup Gailly and Mark Adler. Zlib example_dict_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/example_dict_fuzzer.c.

[35] Jean-loup Gailly and Mark Adler. Zlib example_flush_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/example_flush_fuzzer.c.

[36] Jean-loup Gailly and Mark Adler. Zlib example_large_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/example_large_fuzzer.c.

[37] Jean-loup Gailly and Mark Adler. Zlib example_small_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/example_small_fuzzer.c.

[38] Jean-loup Gailly and Mark Adler. Zlib minigzip_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/minigzip_fuzzer.c.

[39] Jean-loup Gailly and Mark Adler. Zlib uncompress2_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/zlib_uncompress2_fuzzer.cc.

[40] Jean-loup Gailly and Mark Adler. Zlib uncompress_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/zlib/zlib_uncompress_fuzzer.cc.

[41] Dave Gamble. cJSON cjson_read_fuzzer harness. URL: https://github.com/DaveGamble/cJSON/blob/master/fuzzing/cjson_read_fuzzer.c.

[42] Dave Gamble. cJSON: Ultralightweight json parser in ansi c. URL: https://github.com/DaveGamble/cJSON.

[43] Wentao Gao, Van-Thuan Pham, Dongge Liu, Oliver Chang, Toby Murray, and Benjamin I.P. Rubinstein. Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report). In *International Fuzzing Workshop*, FUZZING, 2023.

[44] GEOS contributors. GEOS fuzz_geo2 harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/geos/patch.diff.

[45] GEOS contributors. GEOS coordinate transformation software library, 2021. URL: https://libgeos.org/.

[46] GEOS contributors. GEOS: Tools, 2021. URL: https://libgeos.org/usage/tools/.

[47] GEOS contributors. GEOS: C API Programming, 2024. URL: https://libgeos.org/usage/c_api/.

[48] Harrison Green and Thanassis Avgerinos. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *IEEE/ACM International Conference on Software Engineering*, ICSE, 2022.

[49] Conny Gustafsson. cfile: A C code generator written in Python 3, 2024. URL: https://github.com/cogu/cfile.

[50] Philip Hazel. PCRE2 pcre2_fuzzsupport harness. URL: https://github.com/PCRE2Project/pcre2/blob/master/src/pcre2_fuzzsupport.c.

[51] Philip Hazel. PCRE2: Perl-compatible regular expressions. URL: https://github.com/PCRE2Project/pcre2.

[52] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2021.

[53] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. {FuzzGen}: Automatic fuzzer generation. In *USENIX Security Symposium*, USENIX, 2020.

[54] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. Utopia: Automatic generation of fuzz driver using unit tests. In *IEEE Symposium on Security and Privacy*, Oakland, 2023.

[55] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Network and Distributed System Security Symposium*, NDSS, 2021.

[56] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[57] Johannes Kuhlmann. cgltTF: cgltf_fuzz_main harness. URL: https://github.com/jkuhlmann/cgltf/blob/master/fuzz/main.c.

[58] Johannes Kuhlmann. cglTF: Single-file/stb-style C glTF loader and writer. URL: https://github.com/jkuhlmann/cgltf.

[59] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4), 1992.

[60] Yuwei Liu, Yanhao Wang, Tiffany Bao, Xiangkun Jia, Zheng Zhang, and Purui Su. Afgen: Whole-function fuzzing for applications and libraries. In *IEEE Symposium on Security and Privacy*, Oakland, 2024.

[61] Marti Maria. LCMS it8_load_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/lcms/cmsIT8_load_fuzzer.c.

[62] Marti Maria. LCMS overwrite_transform_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/lcms/cms_overwrite_transform_fuzzer.c.

[63] Marti Maria. LCMS transform_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/lcms/cms_transform_fuzzer.c.

[64] Marti Maria. Little CMS: Little color management system. URL: https://www.littlecms.com/.

[65] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 2010.

[66] Georgy Savidov and Andrey Fedotov. Casr-cluster: Crash clustering for linux applications. In *Ivannikov ISP RAS Open Conference*, ISPRAS, 2021.

[67] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, ATC, 2012.

[68] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *IEEE Cybersecurity Development Conference*, SecDev, 2016.

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

[70] Dustin Spicuzza. cxxheaderparser, 2024. URL: https://github.com/robotpy/cxxheaderparser.

[71] SQLite Constortium. SQLite: C-language library that implements a small, fast, self-contained, high-reliability, full-featured, sql database engine. URL: https://sqlite.org/.

[72] SQLite Constortium. SQLite oss_fuzz harness. URL: https://github.com/sqlite/sqlite/blob/master/test/fuzzcheck.c.

[73] Vsevolod Stakhov. libUCL: ucl_add_string_fuzzer harness. URL: https://github.com/vstakhov/libucl/blob/master/tests/fuzzers/ucl_add_string_fuzzer.c.

[74] Vsevolod Stakhov. libUCL: ucl_msgpack_fuzzer harness. URL: https://github.com/vstakhov/libucl/blob/master/tests/fuzzers/ucl_msgpack_fuzzer.c.

[75] Vsevolod Stakhov. libUCL: Universal configuration library parser. URL: https://github.com/vstakhov/libucl/.

[76] Robert Swiecki. honggfuzz, 2018. URL: http://honggfuzz.com.

[77] Raylib Technologies. RayLib: A simple and easy-to-use library to enjoy videogames programming. URL: https://github.com/raysan5/raylib.

[78] The C-Ares Project. C-Ares: a c library for asynchronous dns requests. URL: https://c-ares.org/.

[79] The C-Ares Project. C-Ares create_query_fuzzer harness. URL: https://github.com/c-ares/c-ares/blob/main/test/ares-test-fuzz-name.c.

[80] The C-Ares Project. C-Ares parse_reply_fuzzer harness. URL: https://github.com/c-ares/c-ares/blob/main/test/ares-test-fuzz.c.

[81] The Clang Team. UndefinedBehaviorSanitizer, 2014. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[82] The HDF Group. HDF5: hdf5_extended_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/hdf5/h5_extended_fuzzer.c.

[83] The HDF Group. HDF5: hdf5_read_fuzzer harness. URL: https://github.com/google/oss-fuzz/blob/master/projects/hdf5/h5_read_fuzzer.c.

[84] The HDF Group. Hierarchical Data Format, version 5. URL: https://github.com/HDFGroup/hdf5.

[85] The TCPDump Group. libPCAP fuzz_both harness. URL: https://www.tcpdump.org/.

[86] The TCPDump Group. libPCAP fuzz_both harness. URL: https://github.com/the-tcpdump-group/libpcap/blob/master/testprogs/fuzz/fuzz_both.c.

[87] The TCPDump Group. libPCAP fuzz_filter harness. URL: https://github.com/the-tcpdump-group/libpcap/blob/master/testprogs/fuzz/fuzz_filter.c.

[88] The TCPDump Group. libPCAP fuzz_pcap harness. URL: https://github.com/the-tcpdump-group/libpcap/blob/master/testprogs/fuzz/fuzz_pcap.c.

[89] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[90] Ladislav Zezula. StormLib: an open-source project that can work with blizzard mpq archives. URL: https://github.com/ladislav-zezula/StormLib.

[91] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie, Xiapu Luo, Xinghua Li, Yang Liu, et al. Automata-guided control-flow-sensitive fuzz driver generation. In *USENIX Security Symposium*, USENIX, 2023.

[92] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. {APICraft}: Fuzz driver generation for closed-source {SDK} libraries. In *USENIX Security Symposium*, USENIX, 2021.