TEPHRA: Principled Discovery of Fuzzer Limitations

Vasil Sarafov, David Markvica, Stefan Brunthaler
μCSRL — CODE Research Institute
Universität der Bundeswehr München, Germany
{vasil.sarafov, david.markvica, brunthaler}@unibw.de

***Abstract*—Fuzz testing has proven effective in discovering non-trivial bugs in complex, real-world systems, with coverage-guided greybox fuzzing being a key contributor to this success. Existing research has largely focused on developing new heuristics to increase code coverage, and current benchmarks measure coverage increase or the number of bugs found. However, there is a notable lack of investigation into programming constructs that systematically hinder or prevent fuzzing heuristics from achieving coverage, commonly referred to as "obstacles" or "roadblocks".**

This work makes two key contributions. First, we introduce TEPHRA, a principled methodology that uses semantics-guided synthesis to generate bug-free programs with diverse obstacles and statistically evaluate a fuzzer's ability to overcome them. Second, we implement TEPHRA-C/C++, a concrete instantiation, and generate 26 different C and C++ obstacles. We use these to empirically evaluate the bypassing abilities of 31 contemporary fuzzers, consuming 37.4 CPU years.

Our analysis reveals limitations in current fuzzing heuristics and uncovers bugs in the fuzzers themselves, including `AFL++`. All evaluated fuzzers struggle with certain obstacles, such as floating-point conditionals and character strings. We also find that signed integers are more challenging than unsigned, and some heuristics are overtuned for 32- and 64-bit types, neglecting 8- and 16-bit integers. Overall, we observe a single difficult semantic construct can significantly degrade a fuzzer's overall performance. Our findings demonstrate TEPHRA's effectiveness and highlight avenues for future research in fuzzing techniques.

***Index Terms*—fuzz testing, coverage-guided, roadblocks, obstacles, benchmarking, program synthesis, benchmark suite, empirical study**

I. INTRODUCTION

Context: Miller et al. introduced fuzzing in 1990 as a low-cost testing method that feeds a Program Under Test (PUT) random inputs and monitors for crashes [1]. Since then, fuzzing has evolved into a principled technique for improving program correctness by complementing verification and validation methods. Coverage-guided fuzzing now functions as a stochastic process that samples the PUT's state space, biased toward uncovering new code or dataflow paths. Due to its stochastic nature, fuzzing benefits from the law of large numbers [2]. In practice, increased sampling—via higher throughput or longer campaigns—raises the likelihood of discovering erroneous states. This progress has culminated in recent years with coverage-guided fuzzers automatically finding bugs in complex real-world systems such as web browsers, databases, and system libraries. Additionally, fuzzing campaigns also produce valuable test inputs (seeds) that support long-term regression testing.

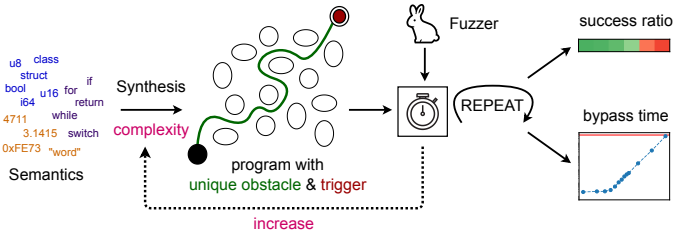

Fig. 1: TEPHRA uses semantics-guided synthesis to generate obstacles and measure fuzzer performance with statistical guarantees.

```
#define MagickEpsilon  1.0e-12
/* ... */
double z = affine->ry * y + affine->tx;
if (fabs(affine->sx) >= MagickEpsilon) {
    /* ... */
    double x = (-z + image->columns) / affine->sx;
    if (x < inverse_edge.x2)
        /* ... */
}
```

Listing 1: Excerpt (simplified) from `ImageMagick`'s `AffineEdge` procedure in `draw.c`, responsible for applying an affine transformation. The two nested comparisons of real numbers exemplify an obstacle blocking a fuzzer to explore additional coverage.

Problem: Unfortunately, the general problem of program validation and bug finding is undecidable [3], [4]. As a result, fuzzers rely on heuristics. That is, a fuzzer is a collection of ad hoc techniques that often work in practice but may fail in certain contexts. Unlike sound and complete formal methods such as abstract interpretation [4], [5], bounded model checking [6], or interactive theorem proving [7], fuzz testing lacks a fundamental theory. Therefore, we rely entirely on empirical data to evaluate its effectiveness and limitations, such as benchmark results and real-world fuzzing outcomes.

Specifically, empirical observations suggest that for every non-trivial PUT, a coverage-guided fuzzer eventually reaches a coverage plateau [8]. At this point, it stops making progress, even though additional coverage is possible. The fuzzer fails to generate inputs that reach unexplored states, and thus misses potential bugs. In other words, the fuzzer gets stuck.

Consider code listing 1, a simplified example extracted from `ImageMagick` [9], a popular open-source image processing suite. It is continuously fuzzed by `oss-fuzz` [10]. Nev-

ertheless, all current fuzzers used in `oss-fuzz` fail to go beyond the chain of comparisons depicted in this listing. These comparisons act as obstacles that block further exploration.

Unfortunately, we lack a systematic understanding and approach for handling such obstacles. In practice, users typically conduct an initial fuzzing campaign, ideally until reaching a coverage plateau, then inspect choke points and attempt to remove them before restarting. They may restart with a different seed, write a new fuzzing driver, construct a grammar to generate structured input, or use in-source intrinsics like `IJON` [11]. While this process can help, it is often tedious, especially for large projects with many choke points, and undermines the core appeal of fuzzing—automation. Furthermore, even when writing a new driver, users do not know how much manual effort is necessary to prevent the fuzzer from getting stuck again.

Intuitively, we cannot expect a fuzzer to bypass *all* obstacles. Consider an obstacle involving, e.g., a `sha256` checksum. Doing so would mean we expect the fuzzer to perform automatic cryptanalysis. Similarly, we cannot expect the fuzzer to solve other inherently hard problems. Clearly, some obstacles are unsolvable and require manual intervention. However, how hard is "too hard" for a fuzzer? Anecdotal evidence suggests that fuzzers often struggle with much simpler obstacles. Where are the actual limits of current coverage-guided heuristics? Are some heuristics more effective than others on specific types of obstacles?

Our proposal: To answer these questions, we introduce TEPHRA, a principled methodology for evaluating a fuzzer's ability to explore hard-to-reach program states. Unlike existing bug-finding benchmarks such as `LAVA` [12], `MAGMA` [13], or `FuzzBench` [8], TEPHRA takes a fundamentally different and complementary approach.

TEPHRA relies on pseudorandom semantics-guided program synthesis to generate obstacle snippets of varying complexity (e.g., Listing 2), and uses an analytical model to measure fuzzer bypassing ability with statistical guarantees (cf. Figure 1). That is, TEPHRA does not comprise a fixed set of benchmarks. Notably, the synthesizer probabilistically expands Programming Language (PL)-specific attribute grammar rules to enumerate different semantic constructs, such as nested predicates, mixed data types, and information flow dependencies (cf. Figure 2). TEPHRA executes each synthesized obstacle multiple times and reports three metrics: 1) the success ratio in bypassing the obstacle, 2) the mean time to bypass it, and 3) a confidence interval to quantify variability in trigger time. A bypass counts as successful iff it is reproducible.

TEPHRA's pseudorandom synthesis and analytical model reduce the risk of overfitting, a common issue in existing fuzzing benchmarks. Its bottom-up, grammar-driven approach enables the generation of diverse obstacles, unlike template-based methods such as `HyperPUT` [14] or `Fuzzle` [15]. This allows TEPHRA to systematically probe the semantic space. Such probing tests a fuzzer's robustness and completeness, as real-world programs often contain unique obstacles and the space of program behaviors is effectively infinite.

Contributions: This work contributes the following:

1) A systematic analysis and categorization of obstacles that hinder fuzzers from exploring complex PUT states.
2) TEPHRA, a principled methodology, based on semantics-guided obstacle synthesis coupled with an analytical measurement model for evaluating a fuzzer's ability to bypass obstacles.
3) TEPHRA-C/C++, a concrete implementation of the generic TEPHRA principle for the C and C++ programming languages. The design of TEPHRA is still PL-agnostic and is readily applicable to other programming languages, such as Java, Rust, or Go.
4) An empirical study, comprising over 37.4 CPU years of computing time, to investigate the limits of 31 different fuzzing systems, using 26 different C and C++ obstacles. We observe limitations stemming from both easy- and hard-to-solve path constraints. Importantly, we find that lack of support for a single programming construct can significantly degrade overall fuzzer performance. In addition, we also find bugs in some fuzzers, most notably `AFL++`. We note the empirical study is not exhaustive w.r.t. the possible obstacles types. Our study applies the TEPHRA methodology, demonstrating its effectiveness and yielding practical insights into real-world fuzzing. TEPHRA's utility extends beyond this evaluation, and we plan more experiments as future work.

Data availability: All TEPHRA source code and data are publicly available at https://ucsrl.de/research/tephra.

II. OBSTACLES: A MODEL

To measure a fuzzer's ability to bypass obstacles, we must first identify what constitutes an obstacle. For example, consider again Listing 1 from `ImageMagick`. In this case, the obstacle consists of two nested `if` statements with predicates between `double` variables that depend directly on fuzzer-controlled input. Intuitively, an obstacle is a condition that prevents a fuzzer from reaching new program states. However, what types of obstacles, i.e., conditions are there?

We aim to identify the semantic constructs that produce challenging code and data patterns, and understand the relationship between program semantics and fuzzing obstacles. To achieve this, we analyze over 50 open-source projects from `oss-fuzz` and identify choke points where current fuzzers struggle. We also conduct a *bottom-up* study of semantic constructs in C and C++. Specifically, we manually examine the C99 and C++11 specifications to identify language features that serve as building blocks for hard-to-reach program states, such as, e.g., `if`, `while`, and exceptions. This analysis yields 14 *high-level obstacle categories* (cf. Table I), an application-oriented definition of an obstacle that is essential for TEPHRA's design, and a discussion that highlights the limitations of capturing obstacles within a formal model.

Simply put, an obstacle is a set of program conditions that no input in the fuzzer's current seed corpus satisfies, preventing the PUT from reaching some new program state:

Definition 1 (Obstacle): Let $I = \{i_1, i_2, ..., i_m\}$ be an initial set of concrete inputs (e.g., a seed corpus). We define the obstacle O w.r.t. I as $O(I) = p_1 \wedge p_2 \wedge ... \wedge p_n$ such that each predicate p_j is false for every $i \in I$.

Based on this definition, two key questions arise: 1) When is an obstacle O considered solved?, and 2) What forms do the individual predicates p_j take?

We consider an obstacle O solved when a fuzzer produces an input that satisfies *all* its predicates p_j. Although multiple inputs may satisfy the obstacle, any single one is sufficient. Inputs that satisfy only a subset of predicates do not count as solutions.

The predicates p resemble path condition predicates from symbolic execution. However, classic symbolic constraints are too fine-grained and impose practical limitations when used as an abstraction for fuzzing heuristics, as we explain below.

A predicate p as used in classic symbolic execution corresponds to a control-flow branch. Furthermore, the program state is defined w.r.t. the semantic model used by the symbolic executor. For machine code, this might be the set of registers and memory cells. For code in high-level programming language, such as C or C++, it may be the set of variables and the call stack.

Unfortunately, in practice, a predicate p may depend on more than just the PUT's code. Real-world programs often run as part of larger systems that influence their semantics. Moreover, a PUT may rely on inputs beyond what the fuzzer has supplied. For example, a Linux user-space program might behave differently depending on disk space, system time, or received POSIX signals.

To account for such effects, formal tools like the `KLEE` symbolic executor often model the PUT's environment (e.g., calls to system libraries such as `libc`). However, formal models cannot capture all aspects of complex systems, or may produce formulas too complex for solvers to handle. This limitation contradicts the core advantage of fuzzers: validating real-world, complex software. Fuzzers often find bugs where semantic models fail to reflect actual behavior.

Therefore, we do not formally constrain the domain of predicates p, and our definition of obstacles remains intentionally high-level and broad. Instead, we focus on an application-oriented classification of obstacle types. Table I summarizes the results, derived from our analysis as described above.

We do not claim the list is exhaustive, but it reflects our best effort. As shown in Section III and Section IV, TEPHRA's design remains flexible and can incorporate additional obstacle types. Our implementation, TEPHRA-C/C++, includes only the first five categories, with the rest reserved for future work (Section IV). We base TEPHRA's design on the results of our obstacle analysis.

TABLE I: High-level obstacle predicate categories identified as part of our analysis.

Obstacle Category	Examples
Control flow	`if`, `while`, `for`, `switch`
Information flow	intertwined inputs, data dependencies
Calling context	procedure-based state machines, indirect calls
Programming-language	type mismatches, exception handlers, iterators
One-way functions	cryptographic checksums, encryption
Self-modifying code	JIT compilers
Nondeterminism	(pseudo/true)random numbers
Concurrency, parallelism	scheduler interleaving, race conditions
Time	time sinks ("hangs"), execution time, date
System resources	memory, disk usage, load average
File system	file size or permission checks
Execution environment	environment variables, signals
Hardware	inline assembly, hardware intrinsics, power
External dependencies	dynamic linking, network access

III. DESIGN

TEPHRA's objective is to evaluate whether a fuzzer can automatically reach program states guarded by specific types of obstacles. To do so effectively, we identify four key requirements that such a methodology must meet:

- G1 generate a *diverse* range of obstacles, from simple to complex;
- G2 accurately *measure* whether the fuzzer can bypass an obstacle and how long it takes, without wasting effort on unrelated code paths;
- G3 account for the inherent *randomness* of fuzz testing;
- G4 minimize the risk of *overfitting* in evaluation results.

TEPHRA satisfies these requirements through a principled approach that combines program synthesis and a statistical measurement model (cf. Figure 1). Specifically, TEPHRA uses a pseudorandom, semantics-guided synthesis to generate microbenchmarks, called samples, each containing exactly *one obstacle*. Each sample is executed multiple times to collect reliable performance data. Additionally, TEPHRA incrementally increases obstacle complexity to identify the point at which a fuzzer fails to make progress, thus exposing its limitations. It also enables statistically rigorous comparisons between fuzzers, providing meaningful performance insights grounded in empirical data.

TEPHRA's design is programming-language agnostic. For clarity, we provide concrete examples from our TEPHRA-C/C++ implementation, detailed in the next section.

A. Anatomy of an Obstacle Sample

Samples are defined by a common framework within three dimensions: 1) input, 2) obstacle, and 3) trigger to signal that the fuzzer has successfully bypassed the obstacle.

Definition 2 (Trigger): A trigger is a code sequence, guarded by an obstacle, which leads the microbenchmark to crash with an error code.

The C procedure `abort` is an example for a valid trigger.

All TEPHRA-C/C++ microbenchmarks are self-contained C (or C++) programs, implementing `libFuzzer`'s interface (cf. Listing 2). As such they receive the input as a function argument. Furthermore, they are bug-free and exit immediately without error if execution reaches a non-triggering state. Each microbenchmark contains exactly one unique obstacle. Every new edge discovered by the fuzzer, if it does not terminate the program, is guaranteed to lead to the trigger. This ensures positive reinforcement for coverage-guided heuristics. By

```
1  int LLVMFuzzerTestInput(const uint8_t *data, size_t n){
2      uint16_t x, y, z, u;
3
4      if (n < 8)
5          return TEPHRA_EXIT_FAILURE;
6
7      memcpy(&x, &data[0], 2);
8      memcpy(&y, &data[2], 2);
9      memcpy(&z, &data[4], 2);
10     memcpy(&u, &data[6], 2);
11
12     if (x == 38951)
13         if (y == 13747)
14             if (z == 54130)
15                 if (u == 7810)
16                     BOOM();
17
18     return TEPHRA_EXIT_SUCCESS;
19 }
```

Listing 2: Example obstacle sample generated by TEPHRA-C/C++'s synthesizer (shortened version). Input is passed as a function argument, the conditions on lines 12–15 are the obstacle, and the trigger is on line 16. Here the obstacle is a simple chain of 16-bit wide unsigned integer comparisons of complexity (depth) 4.

short-circuiting non-relevant paths and reinforcing progress, TEPHRA creates best-case conditions for evaluating a fuzzer's ability to bypass the obstacle. This design ensures that the full benchmarking time measures relevant behavior rather than unrelated exploration. TEPHRA defines a consistent analytical model for its measurements.

B. Measures and Analytical Model

TEPHRA measures if a fuzzer can reliably bypass a given type of obstacle within a time budget T. To that end, TEPHRA executes each synthesized sample with $N \geq 60$ repetitions and returns a triple consisting of:

1) *success ratio* r,
2) arithmetic mean *time-to-trigger* $\bar{t}$, and
3) *confidence interval* $\mathcal{I}_\alpha$ of the time-to-trigger ($\alpha = 0.95$).

We derive the $N \geq 60$ bound such that we can safely apply the central limit theorem and guarantee sound comparisons between measurements.

Definition 3 (Time Budget): Each obstacle sample run is limited to a given wall clock time duration T. If the fuzzer has not found the trigger within T, the run is forcefully terminated.

Definition 4 (Time-to-trigger): The time-to-trigger $t < T$ measures how long a fuzzer needed to successfully hit the trigger. More specifically, let $M < N$ be the number of successful triggers. The mean time-to-trigger is then $\bar{t} = \frac{t_1+t_2+...+t_M}{M}$.

Definition 5 (Success Ratio): The success ratio $r = \frac{M}{N} \in [0, 1]$ is the fraction of runs that have successfully hit the trigger. For a hit to be successful, it is required that a fuzzer provides an input, which can reliably reproduce the hit.

Intuitively, every fuzzing campaign is bounded by the time and memory we can invest. Analogously, TEPHRA requires the time budget T to force termination for runs where the obstacle is too hard to bypass.

Note that for a trigger hit to be successful, TEPHRA validates the fuzzer's seed that caused the crash on the original, non-instrumented microbenchmark. This validation is needed to filter PUT transformations which, e.g., flip branches to bypass obstacles, but generate false-positive trigger hits (e.g., `T-Fuzz` [16]).

The success ratio r summarizes how reliably the fuzzer bypasses the given obstacle. Intuitively:

- if $r = 0$, the fuzzer failed to bypass the obstacle,
- if $r \in (0, 0.5]$, bypassing is unreliable, and mostly by chance,
- if $r \in (0.5, 1]$, bypassing is reliable.

The mean time-to-trigger $\bar{t}$ is an estimator for the expected real time-to-trigger t. However, $\bar{t}$ hides the details of how *precise* the estimator is. Therefore, TEPHRA also reports the time-to-trigger's confidence interval, $\mathcal{I}_\alpha$. We observe:

1) the samples $t_1, t_2, ..., t_M$ are independent and from the same population with mean μ and standard deviation σ,
2) if $0 \leq r \leq 0.5$, $\bar{t}$ does not exist, and $r > 0.5$ implies $M > 30$.

With (1) and (2) we can safely apply the central limit theorem, which assures us that $\bar{t}$ is approximately Gaussian distributed. Thus for $\bar{t}$'s α-confidence interval we obtain: $\mathcal{I}_\alpha = [\bar{t} - z_{1-\frac{\alpha}{2}} \frac{s}{\sqrt{M}}, \bar{t} + z_{1-\frac{\alpha}{2}} \frac{s}{\sqrt{M}}]$, where s is the time-to-trigger's sample standard deviation, and $z_{1-\frac{\alpha}{2}}$ is the tabulated value of the standard unit normal distribution.

C. Comparative Analysis

A primary objective of TEPHRA is to give insights if a given fuzzer `A` is better or worse at bypassing obstacles than another fuzzer `B`. To that end we can use TEPHRA's separate measurements for `A` and `B` and compare them to a given level of statistical significance α. If $r_A \leq 0.5$ or $r_B \leq 0.5$ no sound comparison is possible. Otherwise, we differentiate between two cases:

a) Dependent subjects: First, if `B` is directly derived from `A` (e.g., `B` adds an optimization to `A`), we calculate the differences Δ between the paired observations t. That is, $\Delta = \{t_{A_1} - t_{B_1}, t_{A_2} - t_{B_2}, ..., t_{A_K} - t_{B_K}\}$, where $K = \min(M_A, M_B)$. Then, again due to the central limit theorem and $N \geq 60$, we can assume a normal distribution, and we calculate the α-confidence interval $\mathcal{I}_\alpha$ for Δ. If $0 \in \mathcal{I}_\alpha$, there is no statistically significant difference between `A` and `B` at level α.

b) Independent subjects: Otherwise, if `A` and `B` are independent systems (e.g., different fuzzer implementations), we calculate the difference between the mean times-to-hit: $\bar{t} = \bar{t}_A - \bar{t}_B$. Then for the α-confidence interval we obtain: $\mathcal{I}_\alpha = [\bar{t} - z_{1-\frac{\alpha}{2}} s, \bar{t} + z_{1-\frac{\alpha}{2}} s]$. If $0 \in \mathcal{I}_\alpha$, there is no statistically significant difference between `A` and `B` at level α.

We note that for a correct comparative analysis TEPHRA requires identical testing conditions. That is, both `A` and `B` must be measured on an identical hardware system with identical configurations. Comparing TEPHRA results across different experimental setups is unsound.

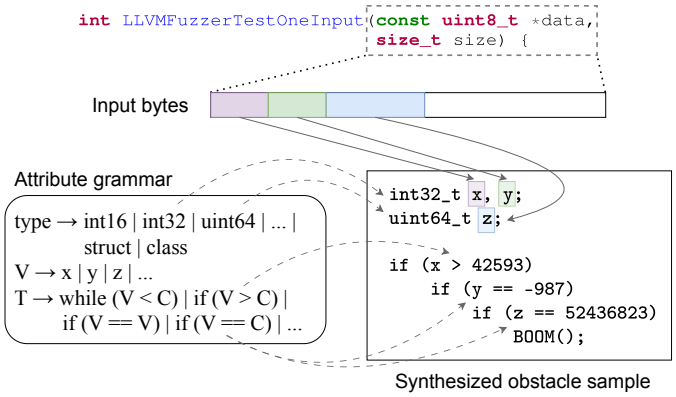

Fig. 2: TEPHRA's synthesis generates obstacle samples by probabilistically expanding PL-specific attribute grammar rules encoding semantic constructs (here C/C++). Each construct corresponds to one obstacle predicate. During synthesis different predicates are chained together to increase the overall complexity.

D. Obstacle Synthesizer

At a high level, the synthesizer probabilistically selects and combines semantic constructs to form an obstacle (cf. Figure 2). To that end, its design is similar to `csmith` [17], and borrows concepts from `xsmith` [18]. However, in contrast to `csmith`, TEPHRA-C/C++ does not attempt to cover the complete semantics of the C programming language. Instead, we opt for a finite number of programming constructs, and use an attribute grammar to drive the bottom-up synthesis. Each programming construct corresponds roughly to one obstacle predicate (cf. Definition 1).

In particular, the synthesizer maintains a variable state similar to a compiler's symbol table. Based on user-defined parameters (e.g., obstacle length), it generates an initial set of variables while tracking each variable's name, type, and its position and length in the input bytestream. Variable types may be primitive (e.g., `char`, `uint32_t`) or composite (e.g., pointer, array, `struct`, or C++ `class`).

The synthesizer outputs a microbenchmark with a unique obstacle. Currently, it generates predicates involving value comparisons, information-flow dependencies, call context, and language-specific features such as C++ exceptions and class methods with dynamic dispatch (see Table I). Obstacles exclude one-way functions, nondeterminism, concurrency, long-running computations, and environment-dependent predicates.

We exclude these obstacle types because most coverage-guided fuzzers cannot automatically bypass them by design, or they are impossible to solve by construction (e.g., cryptography). To our knowledge, only the recent EnvFuzz [19] can handle some environment-dependent predicates. Extending the synthesizer to support additional obstacle types is left for future work.

By default, each generated obstacle depends on every byte of the fuzzer-controlled input. This ensures that fuzzers do not waste effort mutating irrelevant input regions and allows us to evaluate fuzzing heuristics in isolation. However, users may increase the synthesis *breadth* to introduce input-dependent code that does not lead to the trigger. Wider microbenchmarks introduce irrelevant paths, increasing the overall complexity.

IV. IMPLEMENTATION

We implement TEPHRA's principled design with a synthesis for obstacles in C and C++. To that end, TEPHRA-C/C++ consists of three portable command-line applications: an obstacle synthesizer, a benchmark runner, and a tool for comparative analysis. It builds and runs in any POSIX-compliant environment.

a) Obstacle synthesizer: We implement the obstacle synthesizer in 7k lines of Scheme using the CHICKEN Scheme distribution, which compiles to portable C99 and depends only on a POSIX `libc`. It outputs a bug-free, self-contained C (or C++) source file and a pair of input seeds. The source file implements the `LLVMFuzzerTestOneInput` interface (cf. Listing 2), and includes a header that defines the trigger macro `BOOM`. During fuzzing, `BOOM` expands to `abort`, but users may redefine it. During validation, the macro expands to a random cookie, as we explain below.

In addition to an obstacle sample, the synthesizer emits also exactly two well-formed input seeds that the fuzzing process can use as a starting corpus. In particular, these seeds exercise multiple control-flow edges but are guaranteed not to trigger the target condition.

We use a non-deterministic backtracking algorithm to resolve attribute grammar production rules and pseudorandomly generate constants. As a result, the synthesizer is not guaranteed to terminate. To address this, the synthesizer has a restart mechanism that triggers after a user-defined timeout and allows a fixed number of retries. In rare cases where all attempts fail—a scenario we did not observe in practice—the user must adjust the parameters or retry.

The attribute grammar's production rules define the number and type of obstacle predicates to generate (cf. Table I and Definition 1). Some rules are simple, such as comparing a variable to a constant, while others are more complex, such as computing a CRC checksum over a memory buffer (cf. Figure 2). Each predicate provides different degrees of freedom; for example, variable definitions depend on data types, and buffers also vary by size. With TEPHRA-C/C++ one can incrementally increase the obstacle complexity by:

1) adding more predicates to lengthen the conjunction, and
2) increasing the complexity of individual predicates, e.g., with wider data types (e.g., `int8` to `int64`).

We encode constraints in the grammar's attributes to combine predicates through control and dataflow dependencies while ensuring the generated code remains idiomatic C or C++. For example, loop counters are restricted to integer variables. Whether and how predicates are combined depends on the pseudorandomly resolved production rules. Some rules, such as procedure calls, may require generating new variables if the current synthesis state lacks sufficient entries in the symbol table.

Users can control the synthesizer's pseudorandom behavior by setting a seed to ensure reproducibility. They can also choose between normal and exponential distributions and configure their parameters to influence constant values and grammar expansion preferences.

b) Benchmark runner: We implement the benchmark runner in 5k lines of C99. For each trial—defined as a tuple of fuzzer, sample, and repetition ID—the runner creates a dedicated working directory, and monitors memory and time usage, terminating the trial if limits are exceeded.

To support a fuzzer, the runner uses a file-based interface. Each fuzzer provides two executables at a predefined location: `build` and `fuzz`. The `build` script instruments the given sample. The `fuzz` script runs the fuzzer and terminates when it hits the trigger. Its implementation is fuzzer-specific (cf. Table II, last column). The runner communicates with the `build` and `fuzz` scripts via environment variables. We implemented drivers for the 31 fuzzers used in our study (cf. Section V) in 1k lines of POSIX shell.

After completing all trials, the runner collects the results. For each crash-inducing seed a fuzzer has found, the runner validates that the seed triggers the obstacle by redefining the `BOOM` macro to emit a 256-byte random cookie via `printf`. The runner marks a trial as a successful bypass if the non-instrumented sample outputs the cookie. The validation phase does not count against the fuzzing timeout.

c) Comparative analysis tool: We implement TEPHRA's comparative analysis (cf. Section III-C) in 2k lines of C99. It outputs a ranking of the fuzzers at a specified significance level. The TEPHRA distribution also includes 2k lines of Python tools for result visualization and further analysis.

V. EMPIRICAL STUDY

To demonstrate TEPHRA's utility, we use TEPHRA-C/C++ to empirically explore some of the limitations of modern general-purpose fuzzers. Exhaustively evaluating all possible obstacles is prohibitively expensive. Instead, we design our experiments around the following focused research questions:

- RQ1 Are there solvable obstacles that prevent fuzzers from reaching new coverage? If so, what are they?
- RQ2 Do some fuzzing heuristics perform significantly better on certain obstacle types? If so, by how much?
- RQ3 How long does a fuzzer take to bypass a specific type of obstacle?

To answer these questions, we synthesize 26 different obstacles, and group them into two experiment families: isolation (Section V-C) and combination (Section V-D). These obstacles use only the first five predicate categories from Table I.

The first experiment is a *control* experiment, where we evaluate each fuzzer's ability to bypass a *single* obstacle type in isolation (e.g., nested comparisons of one data type). This setup isolates effects other than the obstacle and reveals the causal relationship between a fuzzer's performance and the specific programming construct, with less ambiguity.

In contrast, the second experiment evaluates each fuzzer on obstacles composed of *multiple* programming constructs (e.g., branch-based conditionals with different data types, function call contexts, etc.). This allows us to study how fuzzers handle compound challenges typically found in real-world systems and identify possible synergetic effects. Specifically, if a fuzzer performs well on two isolated obstacle types, it should also perform well on their combination; failure suggests a suboptimal strategy or a new emergent limitation. Conversely, if an isolated obstacle type proves to be challenging for a fuzzer, it should also negatively impact other constructs that a fuzzer would otherwise handle easily.

In the first experiment, all obstacles reduce to arithmetic operations on various data types and their associated conditional control flow. In contrast, the second experiment includes procedure calls, loops, and arrays.

A. System Setup

We conduct experiments on six identical machines with the following specifications: AMD Ryzen Threadripper PRO 7995WX (96 cores, 192 threads), 512 GB DDR5 RAM, and Debian 12.9 (bookworm) with kernel 6.1.0.

Fuzzers run in a `ramdisk`-mounted directory to reduce I/O overhead. Each machine runs up to 94 fuzzers, one pinned per core, with two cores reserved as a buffer.

Measurements run inside a Linux container, initialized from either an Ubuntu 22 or Ubuntu 16 image, depending on the fuzzer's requirements. We implement the container environment in 1k lines of shell and manage it with `Podman` 4.3.1.

B. Methodology

Each obstacle trial runs for up to 1 hour with a 5 GB memory limit. We chose these limits based on the obstacle types and available hardware resources. More concretely, TEPHRA obstacles are not full programs; paths that do not lead to the trigger are short-circuited (see Section III). Furthermore, the obstacle samples are CPU-bound, and we impose no limits on the instrumentation phase. In some cases, however, we observe the instrumentation phase performs poorly. For example, `AFL++` takes 30 minutes to compile samples with more than 4096 nested conditions, while `datAFLow` uses over 10 GB of RAM. The memory and time limits apply only during fuzzing and are primarily consumed by the fuzzer itself.

We repeat each run 60 times. We compute confidence intervals and compare with a significance level of $\alpha = 0.95$.

We evaluate 31 general-purpose fuzzers or variants (cf. Table II) and use the latest version of each at the time of writing. We exclude grammar-based fuzzers, as they are not fully automatic and do not generalize to arbitrary obstacles. Initially we included the `Arvin` [45] fuzzer in our experiments but removed it due to repeated crashes that prevented it from completing. We also evaluated `T-Fuzz` [16] but excluded it because its crash analyzer failed to work.

Our fuzzer selection followed several constraints to ensure diverse and realistic experiments. We included blackbox, greybox, and directed fuzzers, covering a superset of those already used in `MAGMA` and `FuzzBench`. Where available we incorporated relevant configurations that are designed to tackle

TABLE II: Fuzzing systems that are part of our empirical study. The last columns indicates how the trigger support was implemented (○ white-box, ◐ greybox, ● black-box, ⤢ directed, ≈ dataflow, ⌶ binary).

Fuzzer	Type	Derivative of	Version	Release Date	Trigger
AFL GCC [20] (original AFL)	◐	-	2.57b	2020-06-30	AFL_BENCH_UNTIL
AFL clang-fast	◐	AFL	—"—		AFL_BENCH_UNTIL
AFL libtokencap	◐	AFL	—"—		AFL_BENCH_UNTIL
AFL QEMU	◐⌶	AFL	—"—		AFL_BENCH_UNTIL
AFL++ [21]	◐	AFL	4.31c	2025-02-10	AFL_BENCH_UNTIL
AFL++ CmpLog [22], [23]	◐	AFL++	—"—		AFL_BENCH_UNTIL
AFL++ laf-intel [24]	◐	AFL++	—"—		AFL_BENCH_UNTIL
AFL++ MOpt [25]	◐	AFL++	—"—		AFL_BENCH_UNTIL
AFL++ QEMU CmpLog [22]	◐⌶	AFL++	—"—		AFL_BENCH_UNTIL
AFL++ QEMU CompCov [26]	◐⌶	AFL++	—"—		AFL_BENCH_UNTIL
AFLGo [27]	◐⤢	AFL	fa125da	2023-09-28	AFL_BENCH_UNTIL
Angora [28]	◐	-	6b46c85	2022-04-13	custom patch
Angora mb	◐	Angora	—"—		custom patch
Angora random	◐	Angora	—"—		custom patch
DARWIN [29]	◐	AFL	dc51fc4	2023-02-28	AFL_BENCH_UNTIL
datAFLow [30]	◐≈	AFL++	f0e7c25	2023-08-19	AFL_BENCH_UNTIL
DDFuzz [31]	◐	AFL++	319f702	2022-02-28	AFL_BENCH_UNTIL
dev-urandom	●	-	-		own implementation
EcoFuzz [32]	◐	AFL	1fd9460	2020-04-30	AFL_BENCH_UNTIL
FA-Fuzz [33]	◐	AFL	27ac6dc	2023-08-23	AFL_BENCH_UNTIL
FairFuzz [34]	◐	AFL	e529c1f	2019-01-15	AFL_BENCH_UNTIL
honggfuzz [35]	◐	-	defed10	2025-02-25	exit_upon_crash
honggfuzz QEMU	●⌶	honggfuzz	—"—		exit_upon_crash
KLEE [36]	○	-	4928009	2025-05-02	exit-on-error-type
LibAFL [37]	◐	-	f343376	2024-08-15	default
LibAFL_libFuzzer [38]	◐	LibAFL	—"—		default
libFuzzer [39]	◐	-	19.1.7	2025-01-14	reload=0
libFuzzer Entropic [40]	◐	libFuzzer	—"—		reload=0
Radamsa [41]	●	-	653b5ce	2024-02-27	custom patch
SymCC [42]	○	-	37ae14d	2025-03-11	custom patch
WingFuzz [43], [44]	◐	libFuzzer	6ef3281	2022-12-05	reload=0

specific obstacles; for example, `AFL libtokencap` aids with string comparisons, `AFL++ laf-intel` with wide-bit integer comparisons, and `AFL++ CmpLog` with complex comparisons, including floating-point support. In contrast, fuzzers like `honggfuzz` and `Angora` address similar challenges using more holistic approaches, such as gradient-inspired search. Similarly, `QEMU`-backed fuzzers allow us to evaluate the effectiveness of having a machine-code view of the PUT instead of relying solely on high-level source code. We also included both recent fuzzers (e.g., `WingFuzz`) and older ones that score subpar results on existing benchmarks. For instance `FairFuzz`—although considered slow—outperforms `AFL++` on certain obstacle types. Due to resource constraints, we could not integrate every existing fuzzer, particularly from recent work [46]–[48]. However, this is an implementation detail, as our evaluation framework supports easy integration and can accommodate additional systems in future work.

Furthermore, we disable sanitizers to avoid runtime overhead. All fuzzers run in persistent mode or a variant that reduces fork overhead. We explicitly configure the directed fuzzer `AFLGo` to always target the trigger.

We use two baselines to provide reference points for interpreting our results:

1) The custom `dev-urandom` fuzzer outputs data from `/dev/urandom`, simulating pure randomness.
2) The `KLEE` symbolic executor, which uses the `STP solver` to solve path constraints and generate crash-inducing inputs, representing a pure white-box approach.

Each trial uses TEPHRA's default seed corpus (cf. Section IV). For each obstacle type, we increase complexity incrementally until the fuzzer fails at *five* consecutive levels.

After each experiment, we analyze TEPHRA's reports and inspect the success ratio r, the mean time-to-trigger $\bar{t}$, and its confidence interval $\mathcal{I}_{0.95}$. We also monitor the error rate and examine logs for anomalies. For example, some of the more complex synthesized obstacles triggered bugs in the parser and typechecker of upstream `Clang`/`LLVM`. Others exposed problems in the fuzzers themselves, as discussed below.

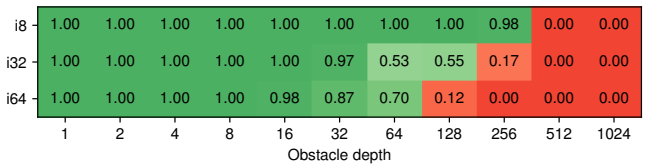

Fig. 3: Heatmap for `libFuzzer`'s success ratio at bypassing the 8-bit, 32-bit, and 64-bit signed integers constant chain obstacles. With increasing obstacle complexity, `libFuzzer`'s performance decreases gradually, i.e., has *stable* performance.

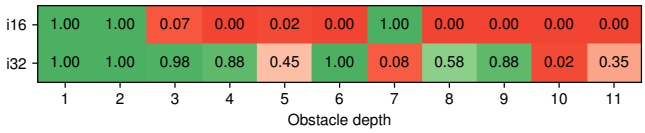

Fig. 4: Heatmap for `EcoFuzz`'s success ratio at bypassing the 16-bit and 32-bit signed integers constant chain obstacles. `EcoFuzz`'s performance is *unstable*. It can fail on easier obstacles, and sporadically solve more complex ones.

We use unstable performance as an additional high-level indicator in our analysis. Ideally, a fuzzer's bypass success should degrade gradually with increasing obstacle complexity (cf. Figure 3). Erratic performance (cf. Figure 4) signals instability, possibly caused by heuristic-sensitive variables not captured by the complexity metric. For instance, some fuzzers may prefer certain integer ranges.

Due to space constraints, we omit the full set of graphs for the 26 obstacle types. Instead, we summarize results in two tables, one per experiment family (Tables III and IV). Listing 2 and Listing 3 show code examples for all obstacle samples except the CRC cases.

C. Isolated Obstacles

```
int LLVMFuzzerTestInput(const uint8_t *data, size_t n){
    float value, eps = 0.0001, bound = 0.07941;

    if (n < sizeof(float))
        return TEPHRA_EXIT_FAILURE;

    memcpy(&value, data, sizeof(float));

    if (value <= (bound + eps))
        if ((bound - eps) <= value)
            BOOM();

    return TEPHRA_EXIT_SUCCESS;
}
```

Listing 3: Example `float`-interval obstacle sample generated by TEPHRA-C/C++'s synthesizer for the isolated experiment family (Table III). Here the obstacle's complexity is determined by the interval's size, which is 10^{-4}.

TABLE III: Results from the isolated obstacles experiment family. Chain obstacles involve nested comparisons of a given data type (cf. Listing 2); complexity is measured by nesting depth (or buffer length for `memcmp`). Higher values are better. Interval obstacles require finding a value near a random point of a given type (cf. Listing 3); complexity is defined by neighborhood width as a power of ten. Lower values are better. Blue shades highlight the top-3 performers per category (from darker to lighter). Underlined yellow entries indicate unstable performance. Orange highlights unstable entries that rank in the top-3.

Fuzzer	Chain													Interval 10^x							
		Unsigned Int				Signed Int				Float		Cmp		Unsigned Int			Signed Int			Float	
	bool	8	16	32	64	8	16	32	64	32	64	str	mem	16	32	64	16	32	64	32	64
AFL GCC	15	3	1	0	0	3	1	0	0	0	0	0	2	0	3	14	0	4	14	−4	−2
AFL clang-fast	14	15	1	0	0	14	1	0	0	0	0	0	2	0	3	14	0	4	14	−4	−2
AFL libtokencap	16	15	1	0	0	15	1	0	0	0	0	1	2	0	3	13	0	4	14	−4	−2
AFL QEMU	32	4	1	0	0	4	1	0	0	0	0	0	2	0	3	14	0	5	14	−4	−1
AFL++	15	15	2	64	32	14	5	3	32	0	0	5	32	0	3	13	0	4	12	−4	−3
AFL++ CmpLog	15	15	0	64	64	14	6	3	64	6	7	64	32	0	3	13	0	4	13	−6	−15
AFL++ laf-intel	15	13	0	16	1	13	1	3	1	1	1	4	16	0	1	11	0	0	11	−2	−1
AFL++ MOpt	15	14	0	32	0	14	5	3	32	0	0	0	32	0	3	13	0	3	12	−4	−3
AFL++ QEMU CmpLog	15	3	2	0	0	3	2	0	0	0	0	0	2	0	3	13	0	4	12	−6	−3
AFL++ QEMU CompCov	16	15	7	3	5	15	7	3	5	0	0	0	8	0	0	0	0	0	0	−4	−3
AFLGo	15	3	2	0	0	2	1	0	0	0	0	0	2	0	3	13	0	4	13	−4	−3
Angora	4096	8192	4096	2048	1024	8192	4096	2048	1024	0	0	0	2	0	1	0	0	0	0	−4	−3
Angora mb	8192	8192	4096	2048	1024	8192	4096	2048	1024	0	0	0	2	0	3	13	0	3	13	−4	−3
Angora random	8192	8192	10	0	0		14	0	0	0	0	0	2	0	4	13	0	4	12	−4	−3
DARWIN	512	64	1	0	0	64	3	0	0	0	0	0	2	0	3	13	0	3	12	−4	−4
datAFLow	14	2	1	0	0	2	1	0	0	0	0	0	2	0	3	13	0	4	14	−4	−3
DDFuzz	512	16	2	4	10	32	1	10	10	0	0	16	32	0	3	13	0	4	12	−4	−3
dev-urandom	13	3	1	0	0	3	1	0	0	0	0	0	3	0	2	12	0	2	12	−6	−4
EcoFuzz	128	14	1	9	0	15	7	9	0	0	0	0	4	0	0	7	0	0	6	−4	−3
FA-Fuzz	512	14	2	0	0	13	1	0	0	0	0	0	2	0	3	13	0	4	13	−4	−3
FairFuzz	1024	128	7	0	0	128	8	0	0	0	0	0	2	0	3	13	0	4	13	−4	−3
honggfuzz	256	32	32	16	16	32	32	32	16	0	0	32	32	0	0	0	0	0	0	−4	−1
honggfuzz QEMU	16	13	6	3	1		7	3	2	0	0	0	8	0	1	13	0	0	8	−4	−1
KLEE	128	128	64	32	16	128	64	32	16	0	0	16	128	0	0	0	0	0	0	0	0
LibAFL	16	15	7	3	16	15	7	10	16	0	0	5	32	0	3	12	0	2	11	−5	−5
LibAFL_libFuzzer	512	64	5	0	1024	64	3	1024	1024	0	0	0	3	0	0	0	0	0	0	−5	−5
libFuzzer	512	256	64	16	128	128	32	64	32	0	0	64	64	0	0	0	0	0	0	−4	−3
libFuzzer Entropic	512	256	32	16	64	256	32	128	64	0	0	64	64	0	0	0	0	0	0	−5	−3
Radamsa	13	2	1	0	0	2	1	0	0	0	0	0	2	–	4	13	–	4	14	−4	–
SymCC	13	9	9	9	10	9	9	9	10	0	0	0	16384	0	0	0	0	0	0	0	0
WingFuzz	1024	256	128	32	128	256	32	256	128	0	0	64	64	0	0	0	0	0	0	−5	−3

Table III shows the results from the isolated obstacles experiment family. Chain obstacle complexity increases linearly from 1 to 16, then exponentially with a factor of 2. Interval obstacle complexity is defined by all powers-of-ten interval lengths allowed by the data type without causing undefined behavior. We observe fuzzers exhibit varying performance across obstacle types, with both strong and weak bypassing results present.

1) Positive results: Some obstacles are easily bypassed by all fuzzers without issue, such as the `u16` and `i16` intervals. In some cases, fuzzers even outperform `KLEE`. For example, in the `strcmp` chain obstacle, `AFL++ CmpLog`, `libFuzzer`, and `WingFuzz` solve up to 64 nested string comparisons (each with length of up to 20 characters), while `KLEE` only reaches 16. Overall, the `libFuzzer` family, including `WingFuzz` in particular, handles all integer intervals and shows stable performance across all obstacle types.

However, some obstacle types remain challenging. In the following, we address each research question and highlight key findings on fuzzer limitations revealed by our results.

2) RQ1: Obstacles hindering fuzzers:

a) Rational numbers: In general, all fuzzers struggle with obstacles involving `float` or `double` as the primary data type. Only `AFL++ CmpLog` solves up to 6 and 7 nested comparisons for `float` and `double`, respectively; all others fail. Fuzzers perform similarly poorly on rational-number intervals, with none outperforming `dev-urandom`. Additionally, we identified two bugs in `AFL++`: one causes a crash, and another leads `AFL++ laf-intel` to produce invalid results. Surprisingly, `KLEE` also lacks proper support for rational numbers and performs worse than most fuzzers on the interval obstacles with rational numbers.

b) Signed and unsigned integers pose different challenges: We observe significant differences in how multiple fuzzer families handle signed versus unsigned integer obstacles. This is evident in both 32-bit chain and interval obstacles. For example, `AFL++ CmpLog` succeeds up to depth 64 for `u32` chains but fails at depth 3 for signed integers. Similar imbalances appear `libFuzzer` and `honggfuzz`.

c) Datatype discrepancies: Interestingly, 16-bit integer obstacles limit fuzzer performance more than their 32- or 64-bit counterparts. This trend appears across multiple fuzzer families, with `LibAFL_libFuzzer` as a notable example. It reaches depth 1024 for signed 32- and 64-bit integers but fails at depth 3 for 16-bit and at depth 64 for 8-bit integers, despite the smaller search spaces. This suggests that fuzzing heuristics are tuned for 32- and 64-bit types while neglecting narrower types. As shown in the composite experiment (Sec-

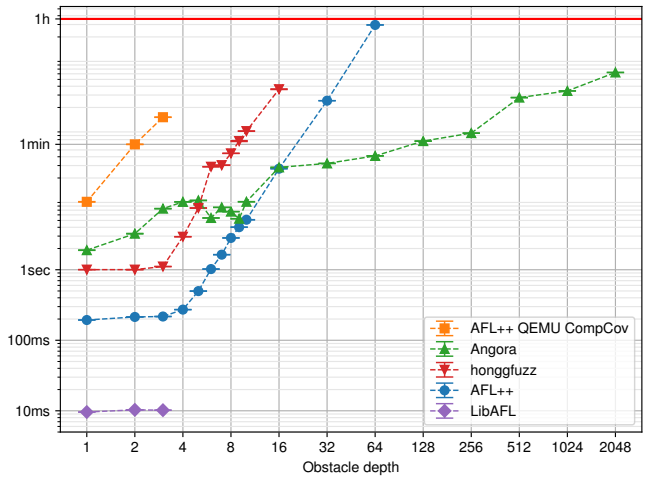

Fig. 5: Performance graph for a subset of the fuzzers' time-to-trigger on the unsigned 32-bit integers constant chain experiment. The bounds for the confidence interval $\mathcal{I}_{0.95}$ are tight.

tion V-D), this bias reduces overall effectiveness and prevents fuzzers from reaching their full potential. We also observe `AFL++ laf-intel`—a fuzzer optimization to specifically handle different integers—performs poorly.

3) RQ2: Fuzzer diversity: We observe substantial variation among the top-performing fuzzers. Some fuzzers excel on specific obstacle types while performing poorly on others. For example, `SymCC` dominates the `memcmp` obstacle, solving inputs of up to 16 KBytes, whereas `KLEE`, the second-best, handles only 128 Bytes. Similarly, `Angora` outperforms all others on integer chains by a large margin and shows stable performance overall. `AFL++ QEMU CompCov` leads among `AFL++` variants on integer interval obstacles. In contrast, `WingFuzz` performs well across all categories. However, no single fuzzer excels at every obstacle type.

4) RQ3: Bypassing time: Due to space constraints, we do not include the full time-to-trigger data. We note fuzzers that solve higher-complexity obstacles within the time limit also tend to be faster. The fastest fuzzers are also the top performers for each obstacle. We highlight two additional observations.

First, time-to-trigger strongly depends on the obstacle type, with fuzzer performance approaching the experiment's time limit at the upper complexity bounds. For example, `Angora` takes 58 minutes to bypass a `u8` chain obstacle of depth 8192.

Second, we observe no significant performance fluctuations. Specifically, 90.62% of all bypasses are reliable. In other words, less than 10% of the bypasses can be attributed to fluctuating randomness. Furthermore, all 0.95-level confidence intervals are tight and remain within 5% of the measured mean time-to-trigger (see, e.g., Figure 5). That is, when a bypass is reliable, the fuzzer consistently triggers it within a narrow time window across repetitions.

D. Composite Obstacles

Table IV presents the results of the composite obstacles experiment family.

Unlike the isolated integer chains, the composite chains mix signed and unsigned integer types. We observe that poor performance on a specific type negatively impacts overall results (cf. Table III). For example, `AFL++` performs well on 32- and 64-bit integers but fails on 8- and 16-bit types, often getting stuck on 16-bit comparisons and failing to reach its full potential. Similarly, weak support for signed integers reduces performance, even when unsigned comparisons are handled well. **In general, we observe insufficient support for a single programming construct degrades overall fuzzer performance, regardless of strengths elsewhere.**

In contrast, `Angora` maintains strong performance across all integer types and preserves its effectiveness from the isolated obstacle experiments.

The CRC obstacles are the most complex in our study. They compute 32- or 64-bit CRC checksums of the input and compare them against an expected answer. In addition to integer arithmetic and boolean conditions, these samples also combine loops, arrays, and function calls.

We observe that no fuzzer outperforms the purely random `dev-urandom` baseline on this task. `KLEE` performs the best and solves CRC constraints for buffers up to 128 Bytes.

TABLE IV: Results from the composite obstacles experiment family. Chain obstacles are nested comparisons on a given data type; complexity is measured by nesting depth. Higher values are better. CRC refers to a sample computing the CRC checksum (32-bit or 64-bit) of the input buffer; complexity increases with buffer length required. Higher values are better.

Fuzzer	Chain				
	Integer			CRC	
	signed	unsigned	all	32	64
`AFL GCC`	1	1	0	2	2
`AFL clang-fast`	1	1	0	2	2
`AFL libtokencap`	1	1	0	2	2
`AFL QEMU`	1	1	0	2	2
`AFL++`	6	7	6	2	2
`AFL++ CmpLog`	16	16	15	2	2
`AFL++ laf-intel`	1	1	1	2	2
`AFL++ MOpt`	4	7	6	2	2
`AFL++ QEMU CmpLog`	1	2	0	2	2
`AFL++ QEMU CompCov`	10	14	13	2	2
`AFLGo`	1	1	0	2	2
`Angora`	2048	2048	2048	2	2
`Angora mb`	2048	2048	2048	2	2
`Angora random`	1	2	0	2	2
`DARWIN`	1	1	0	2	2
`datAFLow`	1	1	0	2	2
`DDFuzz`	3	3	2	2	2
`dev-urandom`	1	1	0	2	2
`EcoFuzz`	7	1	6	2	2
`FA-Fuzz`	1	1	0	2	2
`FairFuzz`	1	2	0	2	2
`honggfuzz`	32	10	16	2	2
`honggfuzz QEMU`	3	4	4	1	1
`KLEE`	16	32	32	128	128
`LibAFL`	16	16	15	2	2
`LibAFL_libFuzzer`	4	7	7	2	2
`libFuzzer`	32	32	16	2	2
`libFuzzer Entropic`	32	32	16	2	2
`Radamsa`	1	1	0	2	1
`SymCC`	9	9	10	0	0
`WingFuzz`	128	64	64	2	2

VI. Discussion

The results of our empirical study enable multiple conclusions and potential applications.

First, knowing what a fuzzer fails to bypass can guide the design of new heuristics or refinement of existing ones, helping to overcome current coverage plateaus. TEPHRA's ability to synthesize isolated obstacles reveals performance issues that are difficult to detect in real-world programs or bug-centric benchmarks. For example, identifying that certain fuzzer families struggle with rational numbers, 8- and 16-bit integers, and that signed integers pose additional challenges, enables more targeted tuning and implementation improvements.

Second, we find TEPHRA's samples act also as stress tests and help uncover bugs in the fuzzers themselves. Ultimately this contributes to more robust fuzzing tools.

Third, finding weaknesses in coverage heuristics and understanding a fuzzer's limits enables users to *predict* where it will struggle. Importantly, this with the creation of fuzz harnesses. The user can focus their effort on hard-to-reach code. It also reduces the need for extensive seed corpus engineering.

When needed, TEPHRA supports statistically sound comparisons between different corpora by measuring their impact on obstacle bypassing. For example, one can evaluate a fuzzer using TEPHRA's default minimalist corpus, repeat the experiment with a carefully crafted alternative, and apply TEPHRA's comparative analysis for dependent subjects (Section III-C).

Our results for RQ2 (fuzzer diversity) indicate that combining multiple fuzzing heuristics can be beneficial. TEPHRA enables fine-grained analysis to identify which heuristics are most effective for specific parts of a PUT, supporting more targeted combinations beyond simple seed sharing (as done in ensemble fuzzing [49]). For example, our results suggest that integrating `Angora` for integer-heavy code with `WingFuzz` for strings and approximate rational values should yield better performance. This approach resembles how SAT solvers switch heuristics at runtime to match problem structure.

Identifying optimal fuzzing techniques is especially relevant in modular frameworks like `LibAFL`, where users assemble fuzzers from libraries of heuristics and each PUT presents a unique set of obstacles. In these cases, TEPHRA results can guide the selection of the most effective components.

Finally, as demonstrated by our use of `KLEE`, TEPHRA is applicable beyond fuzzers and can also support the evaluation of other analyzers, such as concolic and symbolic executors.

VII. Threats to Validity

a) Methodology: The primary threat to TEPHRA's validity is external validity. Since obstacle samples are synthesized rather than extracted from real projects, one may question their relevance to real-world scenarios. However, each obstacle sample represents a minimal, valid semantic constraint that can arise in any PUT. In fact, whether a real-world PUT contains a specific obstacle type depends primarily on its problem domain. For example, nearly all real-world programs contain conditional integer comparisons, making the (un)signed chain obstacles highly relevant. However, only certain PUTs make as extensive use of floating-point operations, as `ImageMagick`.

Moreover, we observe some fuzzers prioritize specific obstacle classes, suggesting that isolating different types of programming constructs is a useful abstraction. Notably, as our results indicate, insufficient support for a single programming construct can degrade overall fuzzer performance. As general-purpose testing tools, fuzzers must either handle arbitrary semantics or clearly indicate when they cannot, allowing users to plan fallback strategies and ensure proper testing. Thus, TEPHRA provides a principled testing ground for evaluating *general-purpose* fuzzers.

Importantly, TEPHRA yields *optimistic upper bounds* on a fuzzer's ability to bypass obstacles. We measure performance under ideal conditions: trigger-irrelevant paths are short-circuited, all input is fuzzer-controlled, obstacles are solvable, no sanitizers are used, and the sample's code is computation-light so that fuzzing remains the bottleneck. Measuring under worst-case conditions would defeat TEPHRA's purpose, which is to reliably expose a fuzzer's limits. If a fuzzer fails under TEPHRA's conditions, it will certainly fail in practice.

b) Implementation: TEPHRA-C/C++ targets user-space C and C++ programs and general-purpose fuzzers. It does not cover domain-specific fuzzers such as syscall or kernel fuzzers, nor does it support firmware or driver fuzzing. It also does not handle language-specific features outside C/C++, such as Rust's safety boundaries. Nonetheless, TEPHRA's insights and overall methodology may benefit domain-specific fuzzers, and adapting the implementation to other languages is feasible.

We also acknowledge the possibility of missing obstacle types. However, the synthesizer is designed to be extensible and can accommodate additional obstacle classes.

c) Empirical study: Our empirical study in Section V remains conservative w.r.t. TEPHRA's capabilities. It does not evaluate more complex obstacles or additional combinations involving information-flow dependencies, function call context, or language-specific features. There is therefore room for further investigation and insights.

Another potential critique concerns our choice of system resources: a 1-hour timeout and 5 GB memory per trial. While the standard fuzzing evaluation protocol typically uses 24 hours and 30 repetitions per PUT [50], [51], TEPHRA samples are standalone snippets that represent components of a PUT. Although more resources can sometimes yield finer-grained results, the configuration we use is sufficient to reveal performance trends. In particular, since fuzzer performance often decreases exponentially with obstacle complexity, a linear increase in resources has limited impact. Furthermore, the limits we choose are enough to observe clear and sometimes orders-of-magnitude large differences between fuzzers.

VIII. Related Work

Due to space constraints, we focus on the most relevant related work and briefly mention others. We also omit discussion already covered in Section I. Specifically, TEPHRA is complementary to top-down bug-finding fuzzing benchmarks such as

LAVA [12], MAGMA [13], Fuzzbench [8], or Unifuzz [52], and the related concept of mutation analysis [53]–[55].

a) Enumeration of program semantics: The principle of enumerating program semantics originates from compiler testing, as compilers must correctly handle all valid programs. TEPHRA follows this approach by synthesizing obstacle samples in a bottom-up fashion using valid semantic constructs. Tools like the csmith family [17], [18] generate C and C++ microbenchmarks to test semantic preservation under optimization, detect missed optimizations [56], [57], and find better optimization sequences [58].

b) Fuzzing obstacles: In a registered report from 2023 [59], Gao et al. argue there is a need to better understand how blockers[1] impede the progress of coverage-guided fuzzers. The authors propose an empirical study to discover blockers in popular and already thoroughly fuzzed open-source libraries (e.g., OpenSSL), using the Fuzz introspector [60] profiling platform. Our work is complementary to Gao et al.'s proposal. We can extend TEPHRA with results from their study once the results are published. Also, with our obstacle analysis (cf. Section II), we attempt to improve our understanding w.r.t. fuzzing obstacles.

Fuzzle [15] generates hard-to-fuzz programs by reducing microbenchmark generation to maze construction, where transitions are encoded as function calls. HyperPUT [14] recently introduced a template-based framework that challenges bug-finding tools using only four transformation templates. Although both use a bottom-up synthesis approach similar to TEPHRA, they differ significantly. First, TEPHRA is a principled methodology that includes obstacle generation and mechanisms to *measure* a fuzzer's effectiveness. Second, TEPHRA supports a broader range of obstacle types and semantic conditions, especially involving data types. For example, TEPHRA can generate obstacles similar to Fuzzle's call-context state machines. Also, unlike template-based systems, TEPHRA does not require manual intervention, allowing more flexible obstacle generation.

c) Benchmarking and testing of program analyzers and solvers: Existing research has focused on testing, benchmarking, and evaluating the quality and robustness of program analyzers [61], [62], such as model checkers [63], [64], and SMT solvers [65], by generating random programs [66]–[68] or building extensive datasets [69]–[72]. In contrast to this research, TEPHRA's microbenchmarks are bug-free and explicitly measure a fuzzer's ability to overcome benign, but non-trivial to analyze program constructs by generating a bypassing seed. Furthermore, TEPHRA can also be used to test program analyzers by confirming that the analyzer finds the path leading to the trigger.

d) Others: Results from TEPHRA support research on the (semi-)automatic generation of *fuzzing drivers*, such as FUDGE [73] and FuzzGen [74]. In contrast to *anti-fuzzing techniques* [75], we make no use of hiding crashes [76], delaying [77]–[79], poisoning [80], or attacking [75] fuzzer-implementation specific properties. However, insights obtained via TEPHRA could be applied for anti-fuzzing.

[1]To the best of our understanding, the authors carry the definition of blockers as defined in Fuzz introspector – "locations in the code where fuzzers are not able to continue execution at run-time despite the code being reachable by the fuzzer based on static analysis."

IX. CONCLUDING REMARKS

Summary: This work investigates obstacles that prevent coverage-guided fuzzers from exploring a program's state space. We introduced TEPHRA, a principled *methodology* to evaluate a fuzzer's ability to bypass diverse obstacles. Importantly, TEPHRA yields *optimistic upper bounds* on a fuzzer's ability to bypass obstacles. TEPHRA measures performance under ideal conditions by, e.g., short-circuiting trigger-irrelevant paths and keeping obstacle code computation-light to ensure fuzzing is the bottleneck. If a fuzzer fails under TEPHRA's conditions, it will certainly fail in practice.

We implemented TEPHRA-C/C++ as a concrete instantiation of the generic TEPHRA principle to synthesize a subset of possible C and C++ obstacle samples and used it to empirically evaluate 31 contemporary fuzzers. Our analysis shows that all evaluated fuzzers struggle with certain semantic constructs, often due to design flaws, implementation gaps, or bugs.

Notably, we find that fuzzers should be improved to better handle 8- and 16-bit integers, as well as rational numbers. Additionally our results indicate that the gap between the difficulty of signed and unsigned integers is large; byte-wise mutations often waste effort on invalid two's complement transformations. Furthermore, we find that subpar support for a single semantic construct, e.g. 8-bit integers, can significantly degrade overall fuzzer performance in complex PUTs. As a result, issues identified using TEPHRA can directly inform improvements in fuzzing real-world targets. Top-performing fuzzers also vary widely, with no single fuzzer excelling across all categories. Finally, we observe current bug-based benchmarks risk overfitting and may not reflect how well a fuzzer generalizes to unseen PUTs. TEPHRA could help fill this gap. We invite the research community to explore our empirical data and use TEPHRA to gain deeper insights.

Future work: We plan to extend TEPHRA-C/C++'s synthesizer and collect more data for additional obstacles. We also aim to explore the predictive power of TEPHRA's results. Specifically, we want to investigate whether we can automatically construct approximate obstacle profiles for real-world PUTs and correlate them with TEPHRA data to pinpoint where a fuzzer will struggle.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and members of the Munich Computer Systems Research Laboratory for their insightful feedback and helpful suggestions. The research was supported by the Austrian ministries BMIMI and BMWET and the State of Upper Austria in the frame of the COMET Module DEPS (FFG 888338).

References

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[2] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS '16)*, 2016, pp. 1032–1043.

[3] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical society*, vol. 74, no. 2, pp. 358–366, 1953.

[4] P. Cousout, "Principles of abstract interpretation," in *MIT Press, 2021*.

[5] P. Cousout and R. Cousout, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGPLAN Symposium on Principles of Programming Languages, Los Angeles, CA, USA, January, 1977 (POPL '77)*.

[6] C. Baier and J.-P. Katoen, "Principles of model checking," in *MIT Press, 2008*.

[7] T. Nipkow, M. Wenzel, and L. C. Paulson, "Isabelle/HOL: a proof assistant for higher-order logic," in *Springer, 2002*.

[8] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM ESEC/FSE*, 2021, pp. 1393–1403.

[9] ImageMagick Studio LLC, "Imagemagick." [Online]. Available: https://imagemagick.org

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

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

[12] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 110–121.

[13] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[14] R. Felici, L. Pozzi, and C. A. Furia, "Hyperput: generating synthetic faulty programs to challenge bug-finding tools," *Empirical Software Engineering*, vol. 29, no. 2, p. 38, 2024.

[15] H. Lee, S. Kim, and S. K. Cha, "Fuzzle: Making a puzzle for fuzzers," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[16] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[17] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[18] W. Hatch, P. Darragh, S. Porncharoenwase, G. Watson, and E. Eide, "Generating conforming programs with xsmith," in *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2023, pp. 86–99.

[19] R. Meng, G. J. Duck, and A. Roychoudhury, "Program environment fuzzing," *arXiv preprint arXiv:2404.13951*, 2024.

[20] M. Zalewski, "American fuzzy lop-whitepaper," *URL: http://lcamtuf.coredump.cx/afl/technical_details.txt*, 2015.

[21] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[22] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *Proceedings of the Networked and Distributed Systems Security Symposium (NDSS)*, vol. 19, 2019, pp. 1–15.

[23] S. Wiebing, T. Rooijakkers, and S. Tesink, "Improving AFL++ CmpLog: Tackling the bottlenecks," ser. Lecture Notes in Networks and Systems. Germany: Springer-Verlag, 2023, pp. 1419–1437.

[24] L.-I. Authors, "LAF-INTEL: Circumventing Fuzzing Roadblocks with Compiler Transformations," August 2016. [Online]. Available: https://lafintel.wordpress.com/

[25] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, Aug. 2019, pp. 1949–1966.

[26] A. Fioraldi, "Compare coverage for AFL++ QEMU," *URL: https://andreafioraldi.github.io/articles/2019/07/20/aflpp-qemu-compcov.html*, 2019.

[27] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.

[28] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[29] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A. Sadeghi, "DARWIN: survival of the fittest fuzzing mutators," *CoRR*, vol. abs/2210.11783, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2210.11783

[30] A. Herrera, M. Payer, and A. L. Hosking, "DatAFLow: Toward a data-flow-guided fuzzer," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–31, 2023.

[31] A. Mantovani, A. Fioraldi, and D. Balzarotti, "Fuzzing with data dependency information," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, 2022, pp. 286–302.

[32] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit," in *Proceedings of the 29th USENIX Security Symposium*, Aug. 2020, pp. 2307–2324.

[33] Z. Gao, H. Xiong, W. Dong, R. Chang, R. Yang, Y. Zhou, and L. Jiang, "Fa-fuzz: A novel scheduling scheme using firefly algorithm for mutation-based fuzzing," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 1–15, 2024.

[34] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 475–485.

[35] R. Swiecki, "Honggfuzz," *URL: https://honggfuzz.dev*, 2016.

[36] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[37] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS '22)*. ACM, November 2022.

[38] A. Crump, A. Fioraldi, D. Maier, and D. Zhang, "LibAFL_LibFuzzer: LibFuzzer on top of LibAFL," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 2023, pp. 70–72.

[39] K. Serebryany, "libfuzzer – a library for coverage-guided fuzz testing," *URL: https://llvm.org/docs/LibFuzzer.html*, 2015.

[40] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting fuzzer efficiency: an information theoretic perspective," in *Proceedings of the 28th ACM ESEC/FSE*, 2020, p. 678–689.

[41] A. Helin, "Radamsa: A general-purpose fuzzer," *URL: https://gitlab.com/akihe/radamsa*, 2016.

[42] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *Proceedings of the 29th USENIX Security Symposium*, Aug. 2020, pp. 181–198.

[43] M. Wang, J. Liang, C. Zhou, Z. Wu, J. Fu, Z. Su, Q. Liao, B. Gu, B. Wu, and Y. Jiang, "Data coverage for guided fuzzing," in *Proceedings of the 33rd USENIX Security Symposium*, Philadelphia, PA, Aug. 2024, pp. 2511–2526.

[44] J. Liang, Z. Wu, J. Fu, Y. Bai, Q. Zhang, and Y. Jiang, "WingFuzz: Implementing continuous fuzzing for DBMSs," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, Santa Clara, CA, Jul. 2024, pp. 479–492.

[45] S. Shahini, M. Zhang, M. Payer, and R. Ricci, "Arvin: Greybox fuzzing using approximate dynamic CFG analysis," in *Proceedings of the 18th ACM ASIA Conference on Computer and Communications Security (AsiaCCS 2023)*, Jul. 2023.

[46] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "{FISHFUZZ}: Catch deeper bugs by throwing larger nets," in *Proceedings of the 32nd USENIX Security Symposium*, 2023, pp. 1343–1360.

[47] D. She, A. Storek, Y. Xie, S. Kweon, P. Srivastava, and S. Jana, "Fox: Coverage-guided fuzzing as online stochastic control," in *Proceedings of the 2024 ACM Conference on Computer and Communications Security (CCS '24)*, 2024.

[48] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "DAFL: Directed grey-box fuzzing guided by data dependency," in *Proceedings of the 32nd USENIX Security Symposium*, 2023, pp. 4931–4948.

[49] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, ser. SEC'19, USA, 2019, p. 1967–1983.

[50] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[51] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 137–137.

[52] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers," in *Proceedings of the 30th USENIX Security Symposium*, 2021, pp. 2777–2794.

[53] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[54] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[55] P. Görz, B. Mathis, K. Hassler, E. Güler, T. Holz, A. Zeller, and R. Gopinath, "Systematic assessment of fuzzers using mutation analysis," in *Proceedings of the 32nd USENIX Security Symposium*, 2023, pp. 4535–4552.

[56] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 697–709.

[57] G. Barany, "Finding missed compiler optimizations by differential testing," in *Proceedings of the 27th international conference on compiler construction*, 2018, pp. 82–92.

[58] A. F. d. Silva, B. N. De Lima, and F. M. Q. Pereira, "Exploring the space of optimization sequences for code-size reduction: insights and tools," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, 2021, pp. 47–58.

[59] W. Gao, V.-T. Pham, D. Liu, O. Chang, T. Murray, and B. I. Rubinstein, "Beyond the coverage plateau: A comprehensive study of fuzz blockers (registered report)," in *Proceedings of the 2nd International Fuzzing Workshop*, 2023, pp. 47–55.

[60] O. Chang, N. Emamdoost, A. Korczynski, and D. Korczynski, "Introducing fuzz introspector, an openssf tool to improve fuzzing coverage," 2022.

[61] D. Kaindlstorfer, A. Isychev, V. Wüstholz, and M. Christakis, "Interrogation testing of program analyzers for soundness and precision issues," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 319–330.

[62] J. Taneja, Z. Liu, and J. Regehr, "Testing static analyses for precision and soundness," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 81–93.

[63] C. Zhang, T. Su, Y. Yan, F. Zhang, G. Pu, and Z. Su, "Finding and understanding bugs in software model checkers," in *Proceedings of the 2019 27th ACM ESEC/FSE*, 2019, pp. 763–773.

[64] M. Pardalos, A. F. Donaldson, E. Morini, L. Pozzi, and J. Wickerson, "Who checks the checkers? automatically finding bugs in c-to-rtl formal equivalence checkers," in *DVCon Europe 2024; Design and Verification Conference and Exhibition Europe*. VDE, 2024, pp. 39–44.

[65] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in smt solvers using blackbox mutational fuzzing," in *Proceedings of the 28th ACM ESEC/FSE*, 2020, pp. 701–712.

[66] M. Fleischmann, D. Kaindlstorfer, A. Isychev, V. Wüstholz, and M. Christakis, "Constraint-based test oracles for program analyzers," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 344–355.

[67] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, "Testing static analyzers with randomly generated programs," in *NASA Formal Methods Symposium*. Springer, 2012, pp. 120–125.

[68] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 203–213.

[69] N. C. for Assured Software, "Juliet c/c++ test suite," October 2017. [Online]. Available: https://samate.nist.gov/SARD/test-suites/112

[70] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5. Chicago, Illinois, 2005.

[71] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "Begbunch: benchmarking for c bug detection tools," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, 2009, pp. 16–20.

[72] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 544–555.

[73] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM ESEC/FSE*, 2019, pp. 975–985.

[74] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 2271–2287.

[75] C. Miller, "Anti-fuzzing," September 2010. [Online]. Available: https://www.scribd.com/document/316851783/anti-fuzzing-pdf

[76] Z. Hu, Y. Hu, and B. Dolan-Gavitt, "Towards deceptive defense in software security with chaff bugs," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 43–55.

[77] E. Edholm and D. Göransson, "Escaping the fuzz - evaluating fuzzing techniques and fooling them with anti-fuzzing," Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016.

[78] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "AntiFuzz: Impeding fuzzing audits of binary executables," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1931–1947.

[79] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1913–1930.

[80] O. Whitehouse, "Introduction to anti-fuzzing: A defence in depth aid," January 2014. [Online]. Available: https://web.archive.org/web/20160310201514/https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/