Combining Input Constraints with Execution Goals

Leon Bettscheider
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
leon.bettscheider@cispa.de

Marius Smytzek
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
marius.smytzek@cispa.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
andreas.zeller@cispa.de

***Abstract*—Recent advances in blackbox test generation (fuzzing) allow combining syntax specifications (grammars) with *semantic constraints* over grammar elements. These constraints not only help in producing valid inputs ("⟨*checksum*⟩ should be a Luhn checksum over ⟨*number*⟩") but also allow specifying *testing goals* ("⟨*payload*⟩ should be as short as possible"). Such constraints give testers unprecedented control over the generated inputs. However, constraint-based fuzzers like ISLa or Fandango have only been limited to *input features* and blackbox fuzzing.**

This paper shows how to extend this concept with *execution constraints* relating to the program under test. We introduce a set of *primitives* that check for specific execution features such as coverage of specific locations, memory usage, execution time, and more. Testers can then use these to specify additional testing goals: "I want an input that is (1) valid, (2) as short as possible, and (3) results in a maximum of code coverage."

We implement our approach on top of the Fandango fuzzer and find that its evolutionary algorithm efficiently finds solutions that satisfy input and execution constraints. We demonstrate that combining input and execution constraints enables us to specify and implement approaches such as directed, performance, and coverage-guided fuzzing, effectively covering the entire range of testing goals from whitebox to blackbox fuzzing in a unified framework.

***Index Terms*—software test generation, fuzzing, grammars, input constraints, execution constraints, greybox testing, blackbox testing, directed fuzzing, targeted testing**

I. Introduction

Traditionally, the world of software testing has been split into two camps. *Blackbox tests* are derived from a *specification,* which exists independently of the program under test (**PUT**); the aim is to ensure that all aspects of the specification are covered. *Whitebox tests* are derived from the *implementation,* which is known to the tester; here, the aim is to cover the *implemented* features. Both paradigms are needed, particularly as an implementation may not fully satisfy its specification, and an abstract specification may not cover all concrete details.

As it comes to *generating* test inputs automatically, though, the vast majority of approaches focus on the whitebox paradigm. That is because the PUT is (1) always available by definition, and (2) comes in a form that (at least in principle) allows analysis of the program's code and execution. This concept contrasts with blackbox test generation, which needs a formal specification of the program's input language. While *context-free grammars* are a common and proven way to specify input languages, they are not expressive enough to capture all aspects of a program's input—notably, *semantic* aspects such as checksums, length constraints, references, and other properties that are non-context-free.

In recent years, a new generation of blackbox test generation tools has emerged that allows users to specify *semantic constraints* over the grammar. These constraints come as logical formulas over grammar elements. In the grammar

$$\langle \textit{start} \rangle ::= \langle \textit{credit-card} \rangle \tag{1}$$

$$\langle \textit{credit-card} \rangle ::= \langle \textit{number} \rangle \langle \textit{checksum} \rangle \tag{2}$$

an input constraint such as

$$\langle \textit{checksum} \rangle = \text{luhn}\big(\langle \textit{number} \rangle\big) \wedge \big|\langle \textit{number} \rangle\big| < 20 \tag{3}$$

expresses that the ⟨*checksum*⟩ digit should be a Luhn checksum over ⟨*number*⟩, and the remaining ⟨*number*⟩ should have fewer than 20 characters. Test generators such as ISLa [1] or Fandango [2] allow users to specify such constraints, and then generate inputs that satisfy them.

Such constraints are useful for generating valid inputs and specifying *testing goals* over the generated inputs. If a tester wants to test *Visa* cards, for example, a constraint such as ⟨*credit-card*⟩.startswith('4') ensures the number starts with a 4; likewise, a constraint such as $\big|\langle \textit{credit-card} \rangle\big| = 2$ will put some length checks to the test. In Fandango, testers can use any Python function to specify such constraints, giving them considerable flexibility in specifying the input language. To satisfy the constraints and produce inputs, Fandango uses robust constraint-aware evolutionary algorithms.

While most whitebox test generators uniquely focus on *increasing coverage* in the PUT, some tools also provide flexibility in specifying testing goals. Specialized fuzzers such as SlowFuzz [3] and PerfFuzz [4] focus on performance-related goals rather than code coverage, such as maximizing program execution time. FuzzFactory [5] provides a framework for composing multiple domain-specific fuzzing strategies on top of AFL. **However, these tools do not allow control of the input language; they do not allow specifying *valid* inputs, and they cannot use the tester's *domain knowledge* on where the bugs might be.**

This work is funded by the European Union (ERC S3, 101093186). Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

In this paper, we propose a new approach that combines the best of the blackbox and whitebox worlds, empowering testers to *specify both input constraints and execution goals in a single framework*. Specifically, we extend Fandango with *additional constraint primitives* that refer to the execution domain, thus allowing an arbitrary mix of input constraints and execution goals. This integration enables the use of greybox fuzzing techniques—including coverage-guided fuzzing, domain-specific feedback-driven fuzzing, and combinations thereof—covering the entire range of approaches from blackbox to whitebox fuzzing in a single framework.

Our approach reuses the evolutionary algorithm of Fandango in a two-phase process: it first generates inputs that satisfy user-defined input constraints, and then *evolves* those valid inputs to optimize execution goals based on feedback from the PUT. Users can thus declare and arbitrarily combine input constraints and execution goals using the complete flexibility of Python.

Let us illustrate this with a simple example. Let us assume we want to determine which credit card number takes the longest to process for validity. With our approach, we can express this using the grammar in Eqs. (1) and (2), the input constraint in Eq. (3), and the execution goal:

$$\textbf{maximizing } \text{DynamicAnalysis}\big(\langle \textit{start} \rangle\big).\text{ExecPathLen}() \quad (4)$$

Here, the keyword **maximizing** indicates a *soft constraint*—a value we want to maximize. *DynamicAnalysis* is a primitive that collects execution feedback from the PUT; Overall, this goal states that we want an input that satisfies the checksum constraint and maximizes the execution path length. Since long executions tend to be trivially produced by long inputs, we could also add another execution goal that we want the input to be as short as possible:

$$\textbf{minimizing } \big|\langle \textit{start} \rangle\big| \quad (5)$$

We argue that such specifications are accessible even to non-expert users; they are far more expressive, readable, maintainable, and combinable than the specialized code of existing domain-specific fuzzers. We demonstrate the practicality and versatility of our approach through a series of experiments on real-world programs implemented in C and C++. Specifically, we implement five domain-specific fuzzing scenarios, including directed, performance, and coverage-guided fuzzing. These goals can be expressed in our unified framework with one to two lines of specification and efficiently solved by the Fandango evolutionary algorithm.

In summary, our main contributions are:

1) We present a novel unified approach that allows *specifying execution goals in conjunction with input constraints*, enabling rich greybox testing applications within the Fandango framework.
2) We integrate *execution feedback capabilities* into the Fandango tool, interfacing with the instrumented PUT and allowing Fandango to optimize non-normalized feedback values. We release this extension as open-source software.
3) We provide a *catalog of domain-specific execution goals,* including *code coverage*, *function coverage*, *basic-block distance*, *execution path length*, and *heap memory usage*.
4) We present and open-source our *LLVM-based instrumentation and analysis passes,* wrapped in a compiler replacement called `fcc`. This tool outputs instrumented binaries and static analysis information as JSON files.
5) We show that our approach can evolve and produce individual *inputs* as well as *populations* that fulfill a particular goal, as required by test settings.
6) We demonstrate the effectiveness of our approach through five domain-specific *fuzzing scenarios,* showcasing its versatility and practicality.

The remainder of this paper is structured as follows: Section II reviews key concepts in language-based testing and modular fuzzing. Section III introduces our unified approach, FANDANGOGREY, combining language-based testing with execution feedback. Section IV describes the implementation of FANDANGOGREY, including our LLVM-based instrumentation framework and the extensions made to Fandango. Section V showcases five domain-specific fuzzing scenarios, demonstrating the practicality and versatility of our approach on real-world programs, followed by potential threats to the validity of our results in Section VI. Section VII presents an outlook on future directions, before we conclude in Section VIII.

II. BACKGROUND

A. Language-Based Software Testing

Language-based software testing [6] has been proposed as a new approach for generating test inputs that satisfy *grammars* (for specifying input syntax) and *constraints* (for specifying input semantics) to define input languages. The first implementation of this methodology is the ISLa framework [1], using SMT solvers to satisfy the constraints. Although this enabled new capabilities in small-scale settings, it did not generalize well across many real-world input formats, primarily because numerous constraints could not be expressed within the solver's limited SMT-LIB language.

The recent Fandango tool [2] shares the same goal as ISLa but is significantly more flexible in terms of the input constraints it supports—in fact, it allows users to specify *arbitrary semantic constraints* using the full expressiveness of Python. Its key innovation lies in employing *search-based testing*—an algorithm that evolves inputs towards satisfying increasingly more specified input constraints, retaining and mutating the *fittest* candidates.

This approach implies that input generation only needs *checking whether input candidates satisfy the constraints*, rather than requiring them to be expressed in the rigid language supported by SMT solvers. Some constraints, such as checksums, are difficult to satisfy through such a *generate-and-check* approach. Fandango, therefore, also allows attaching imperative generator functions to specific grammar non-terminals. For example, a generator might compute a checksum for one non-terminal and insert the result into another.

Despite these advancements, a key limitation remains: As a black-box technique, Fandango cannot incorporate *execution feedback* into the evolution of inputs, which is in contrast to state-of-the-art greybox fuzzers, which build their success on evolving inputs based on execution feedback such as code coverage. Most greybox fuzzers, however, are unaware of input specifications and thus apply mutations that often violate the input format, reducing efficiency and preventing users from expressing testing goals within the *input domain*. We aim to bridge this gap by enabling Fandango to leverage *execution feedback* during input evolution, as detailed in Section III.

B. Combining Input Specifications with Execution Feedback

The idea of combining input specifications with execution feedback is not new. Seminal approaches include *grammar-based whitebox fuzzing* [7], which enhances symbolic execution with context-free grammars; AFLSMART [8], which combines greybox fuzzing with *Peach Pit* specifications; and NAUTILUS [9], which integrates grammars and coverage-guided fuzzing. However, all these approaches are *monolithic*—limited to increasing code coverage as their primary goal and constrained to syntactic specifications—and semantic input constraints and alternative objectives are not supported.

C. Modular Fuzzers

FUZZFACTORY [5] is the first modular AFL-based [10] framework that enables users to concisely implement, reuse, and combine specialized fuzzing strategies—such as *performance* [4], [3] or *validity fuzzing* [11]—in a unified framework. However, it still requires deep knowledge of AFL internals and C++ programming. Moreover, while flexible regarding execution feedback, it does not allow users to incorporate domain-specific input specifications into the fuzzing process.

Similarly, LIBAFL [12], a framework for modular fuzzer development, supports the composition of multiple fuzzers such as NAUTILUS or GRIMOIRE [13]. However, it lacks support for specifying semantic input constraints as well, and using or extending the framework requires a deep understanding of its internals.

Our work extends this idea of modular fuzzing by allowing users *to specify input constraints and execution goals in a declarative and reusable way*, as shown in the next section.

III. FANDANGOGREY: EVOLUTIONARY CONSTRAINT SOLVING MEETS GREYBOX TESTING

Our goal is to extend Fandango with execution feedback, allowing it to generate inputs that not only satisfy user-defined input constraints but also *optimize execution objectives.*

This approach will allow us to specify execution goals such as:

- minimizing the distance to a specific code location,
- maximizing the number of executed basic blocks,
- maximizing the amount of heap memory allocated.

Initially, Fandango only supported **hard constraints**, which are input constraints that *must* be satisfied. For instance, validity constraints ensure that generated inputs conform to the input language—a non-negotiable requirement. In addition, we introduce **soft constraints**, which are not strictly required but represent *desirable properties* that can be optimized. Let us detail how we integrate soft constraints into the Fandango evolutionary algorithm, followed by a description of how we normalize input fitness with respect to these soft constraints.

A. Integrating Soft Constraints

Each input is represented as a *derivation tree*, i.e., a parse tree whose internal nodes are labeled with grammar non-terminals and whose leaves concatenate to form the concrete input string. Fandango defines the fitness of an input as a function $f\colon T \to [0,1]$, which maps a derivation tree $t \in T$ to the average satisfaction score across all input constraints:

$$f(t) = f_H(t) = \frac{1}{|C_H|} \sum_{c \in C_H} s_H(c,t)$$

Here, $C = C_H$ denotes the set of constraints, consisting solely of hard constraints, and $s_H(c,t) \in [0,1]$ represents the satisfaction score of tree t concerning constraint c.

We extend this definition to incorporate *soft constraints*. The overall constraint set is now partitioned into two disjoint subsets: hard constraints C_H and soft constraints C_S, such that $C = C_H \cup C_S$ and $C_H \cap C_S = \emptyset$. Regarding the evolutionary algorithm, the general idea is to structure the search such that Fandango prioritizes generating inputs that satisfy all hard constraints. Subsequently, those inputs are evolved to optimize the soft constraints. To support this, we define a soft constraint fitness function $f_S\colon T \to [0,1]$, which is only evaluated when all hard constraints are satisfied:

$$f_S(t) = \begin{cases} \frac{1}{|C_S|} \sum_{c \in C_S} s_S(c,t), & \text{if } f_H(t) = 1.0 \\ 0, & \text{otherwise} \end{cases}$$

The overall fitness function then becomes an average of the hard and soft components:

$$f(t) = \frac{|C_H| \cdot f_H(t) + |C_S| \cdot f_S(t)}{|C|}$$

Note that $f(t)$ only evaluates to 1 if both $f_H(t)$ and $f_S(t)$ are equal to 1. Clearly, $f_H(t) = 1$ implies that all hard constraints are satisfied, but how do we interpret the value of $f_S(t)$? We will discuss this in the next section.

B. Normalizing Soft Constraints

We define soft constraints as arbitrary Python expressions over non-terminals in the grammar, each accompanied by an optimization goal—either `maximizing` or `minimizing`. For instance, the following soft constraint guides the input generation process towards producing increasingly larger values for the ⟨*age*⟩ non-terminal:

maximizing int(⟨*age*⟩)

Beyond such properties of the *input domain*, soft constraints can also target execution goals, which are measured by running the PUT on the generated input. However, such execution goals are not naturally *normalized.* For instance, we generally

do not know the maximum possible $\langle age \rangle$ value in the context of a larger input specification, the theoretical maximum number of basic blocks that can be covered, or the minimum attainable control-flow distance. As a result, raw fitness values cannot be directly normalized.

To overcome this, we normalize the fitness of inputs regarding soft constraints using a statistical technique based on the *t-digest* data structure [14]. A t-digest maintains a compact summary of a stream of numeric values, enabling efficient quantile estimation without storing the entire history. Importantly, it provides a cumulative distribution function (CDF) that maps a value x to the proportion of prior observations less than or equal to x, i.e., a value in the range $[0, 1]$.

This CDF enables normalization: a high value of $\text{CDF}(x)$ indicates that x is large relative to past observations (favorable for maximization). In contrast, a low $\text{CDF}(x)$ value suggests that x is small (favorable for minimization). We further apply exponential scaling to the CDF to accentuate improvements near the target boundaries—i.e., upper bounds when maximizing and lower bounds when minimizing.

The implementation of our soft constraint score computation is shown in Algorithm 1. We first compute the absolute fitness of the input t concerning the soft constraint c by evaluating the Python expression associated with c in the context of the input (Line 2). We then invoke CDF on the absolute fitness value and store the result in *cdf* before the absolute fitness value is added to the t-digest (Lines 3-4). Finally, based on the optimization goal, we apply exponential scaling to *cdf* and return this normalized value (Lines 5–10).

Algorithm 1 Soft Constraint Score Computation

```
1:  function s_S(c, t, contrast ← 10.0)
2:      absFitness ← c.FITNESS(t)
3:      cdf ← c.tdigest.CDF(absFitness)
4:      c.tdigest.UPDATE(absFitness)
5:      if c.optimizationGoal = "maximize" then
6:          return exp(contrast × (cdf − 1))
7:      else
8:          return 1 − exp(−contrast × cdf)
9:      end if
10: end function
```

IV. Implementation

A. Instrumentation and Program Analysis

We implement a lightweight LLVM-based instrumentation framework that enables the collection of execution feedback from the PUT. With usability in mind, our framework makes it easy to compile existing C/C++ projects with instrumentation. Our instrumentation pass is bundled in a tool called `fcc`, which acts as a drop-in replacement for `clang`. This tool allows users to instrument arbitrary C/C++ programs with minimal effort. For example, a typical compilation with fcc looks like:

```
$ fcc -o f *.c
```

The resulting instrumented binary can then be passed to Fandango, which will execute it and collect runtime feedback.

In addition, we provide a companion tool called `fan`, which performs simple static analysis on `fcc`-compiled programs. It generates an interprocedural control-flow graph enriched with source code metadata such as module names, function names, and line numbers. Fandango uses this information to support execution metrics like `DistanceToFunction()` and `DistanceToLine()`.

`fan` is invoked automatically by Fandango, but users can also run it manually to inspect the static analysis output:

```
$ fan f
```

From a technical perspective, `fcc` builds on top of the `wllvm`[1] tool. `wllvm` compiles once to produce standard object files and again to generate corresponding LLVM bitcode object files. During the linking stage, `wllvm` embeds references to these bitcode objects within the final binary.

To recover the LLVM bitcode for the entire program, we use the `extract-bc` tool provided by `wllvm`, wrapped inside our `fan` tool. `extract-bc` extracts the embedded bitcode object files, which are then linked using `llvm-link` into a single whole-program LLVM bitcode file. This unified bitcode is used as the input for our analysis pass, from which we derive the inter-procedural control-flow graph of the PUT.

Let us describe the instrumentation pass (used by `fcc`) and the analysis pass (used by `fan`) in more detail. Figure 1 illustrates the overall workflow of `fcc` and `fan`.

① **Instrumentation Pass:** This module-level pass assigns each basic block a unique ID and annotates it with this ID and the module name. Each module maintains a map of 64-bit counters indexed by these IDs. When a basic block BB_i in a module is executed, the counter at position i in the module-level map is incremented, ensuring unique identification after linking via module-scoping.
Finally, we register an `atexit()` hook that serializes the execution trace—including covered basic blocks and hit counts—to disk upon program termination.

② **Analysis Pass:** The analysis pass operates on the whole-program bitcode of the PUT. It traverses all functions and their basic blocks to construct an interprocedural control-flow graph, linking individual intraprocedural control-flow graphs via direct function calls. In addition, it enriches each basic block with source code-level metadata, including the module name, function name, and line number. This information enables Fandango to express execution metrics in a clear and user-friendly manner.

Advanced users can extend these LLVM passes to provide Fandango with additional analysis information.

B. Fandango Integration

We integrate our approach into Fandango through two key modifications. First, we extend Fandango to support *soft constraints*, as described in Section III. Second, we introduce an interface to connect Fandango with a user-supplied PUT, instrumented using *fcc*.

[1] https://github.com/travitch/whole-program-llvm

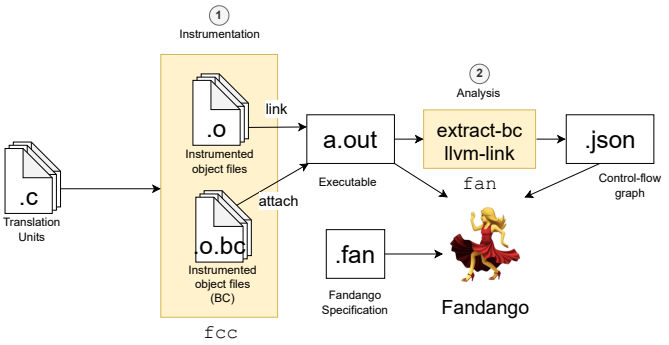

Fig. 1: Instrumentation workflow with `fcc`. Translation units are compiled twice (with instrumentation) using `wllvm`, producing standard and LLVM bitcode object files. These are bundled into the executable, which Fandango runs to generate execution traces. Our `fan` tool extracts static analysis information from the bitcode object files, enabling the computation of execution metrics (Table I).

When the input specification includes the *DynamicAnalysis* keyword, our Fandango extension begins by loading static analysis information extracted from the PUT using the *fan* tool. This information is then processed to enable the use of our execution metrics (Table I). Specifically, control-flow graphs are used to precompute control-flow distances for directed fuzzing. At the same time, debug information is used to map source-level constructs to basic blocks, facilitating more intuitive interpretations of execution metrics.

After this preprocessing step, Fandango executes the PUT on each input that satisfies all hard constraints. For each execution, it reads the execution trace produced by the instrumentation and evaluates the specified execution metric. To improve performance, trace data is cached using a *least-recently-used* (LRU) strategy.

V. Domain-Specific Fuzzing Applications

In Section V-A through Section V-E, we demonstrate the practicality, usability, and versatility of FandangoGrey by applying it on five different domain-specific fuzzing scenarios, each targeting a different execution objective. Some of these domains have previously been explored in isolation by other researchers. However, our goal is not to outperform existing fuzzers in speed—an unfair comparison given that our implementation is in Python. In contrast, most state-of-the-art fuzzers are highly optimized and implemented in C or C++. Instead, our key contribution lies in being *the first to integrate execution goals with input constraints in a unified framework.* Our evaluation thus focuses on demonstrating that the combination of an input specification and execution feedback outperforms either component used alone in Fandango. Our catalog of execution metrics currently comprises those presented in Table I, exploring the following five domains:

(A) **Performance Fuzzing**. In this domain, we aim to find inputs that are both small and slow to execute. We demonstrate this by implementing the `SlowFuzz` technique [3].

(B) **Excessive Memory Fuzzing.** In line with the "mem" application of FuzzFactory [5], we aim to find inputs that allocate excessive amounts of heap memory.

(C) **Maximizing Code Coverage.** We aim to find inputs that maximize the number of basic blocks covered by a single input rather than maximizing coverage across the entire input population, as is typical in AFL-style fuzzers.

(D) **Directed Greybox Fuzzing.** In this domain, we implement a technique similar to "Directed Greybox Fuzzing" [15], which aims to find inputs that reach a specific target location in the program.

(E) **Evolving Populations towards Coverage.** We also implement fuzzing in the style of AFL [10], evolving *populations of inputs* towards maximizing cumulative code coverage—while taking input constraints into account.

We evaluate these domain-specific fuzzing scenarios by running experiments on real-world programs and corresponding input specifications, which we detail in the next section. Each experiment is run for one hour. We repeat each experiment ten times to account for the randomness inherent in fuzzing. After collecting the data from all ten runs, we compute the average and 95% confidence intervals. In our plots, the averages are represented by solid lines, with the surrounding shaded areas indicating the 95% confidence intervals. Each plot compares three different fuzzing setups:

(1) **Execution feedback only:** This setup relies solely on execution feedback to guide fuzzing, using a trivial input specification that can generate arbitrary strings:

$$\langle start \rangle ::= \langle byte \rangle^+$$

(2) **Input specification only:** This setup uses a detailed input specification—consisting of a context-free grammar and input constraints—but does not entail execution feedback.

(3) **Combined approach:** This setup integrates both strategies by combining the execution feedback from setup (1) with the input specification used in setup (2).

The research questions we address in this evaluation are:

1) *Can we combine input constraints and execution feedback concisely?*
2) *Does the combination yield better results than each component alone?*

Note that we do not directly compare against existing fuzzers. For one, fuzzers specialized for one task typically are *optimized* for this very task, at the expense of generality (which we strive for). Also, their *requirements* vastly differ; AFL performance, for instance, largely depends on the quality of the initial seed and the AFL dictionary, whereas the FandangoGrey performance depends on the quality of the input specification and the given constraints, making a fair comparison impossible. The contributions of this work focus on control and versatility, and our evaluation is set accordingly.

Evaluation Subjects

We evaluate our approach on four real-world programs written in C and C++. These programs were selected due to their complex and structured input languages, the availability

TABLE I: Execution Metrics provided by FANDANGOGREY

Fuzzing Goal	FANDANGOGREY Metric	Description
Directed Greybox Fuzzing [15]	`DistanceToFunction(m, f)`	Call distance to function `f` in module `m`.
	`DistanceToLine(m, l)`	Basic block-level distance to line `l` in module `m`.
	`DistanceToBB(m, bb)`	Basic block-level distance to basic block `bb` in module `m`.
Coverage-Based Fuzzing [10], [16], [17]	`FunctionCoverage()`	Number of covered functions.
	`CodeCoverage()`	Number of covered basic blocks.
	`CoverageInFunction(f)`	Number of covered basic blocks in function `f`.
Performance Fuzzing [3], [4], [5], [18]	`ExecutionPathLength()`	Execution path length measured in basic block hits.
	`PeakBasicBlockHits()`	Number of hits of the most frequently executed basic block.
Excessive Memory Fuzzing [5]	`HeapAllocatedBytes()`	Total size of heap memory allocated.

of language specifications, and their prior use in research. Three of the programs—*graphviz* [19], *lunasvg* [20], and *libxml2* [21]—were also used by a recent paper on finding slow but short inputs using grammar-based search [18]. We use these programs in Section V-A—Section V-C and Section V-E. We further evaluate *cjson* [22], a widely used JSON parser implemented in C, in a directed greybox fuzzing scenario in Section V-D. In the following, we briefly describe these programs and their input languages.

graphviz [19] is a popular graph visualization software that primarily uses the DOT language for describing graphs. The DOT language has a text-based syntax and allows users to define nodes, edges, and attributes in a structured manner. We use the exact syntax specification as in [18], which was extracted from the Bison EBNF grammar that ships with the graphviz source code. While graphviz contains multiple tools for different purposes, our experiments focus on the `dot` tool, which generates visual representations of graphs.

lunasvg [20] is a C++ library for rendering SVG (Scalable Vector Graphics) files. SVG is an XML-based vector image format that describes images using geometric shapes, paths, and text. We reuse the input grammar provided as part of the reproducibility package of [18], which was derived from the W3C documentation[2] and focuses on the path element. lunasvg ships with a command-line tool called `svg2png`, which converts SVG files to PNG images. We use this tool in our experiments.

libxml2 [21] is a C library for parsing XML documents, initially developed for the GNOME project. libxml2 is often used in fuzzing benchmarks [18], [5]. We use the XML specification provided in the reproducibility package of Fandango, which includes both a syntax specification and semantic input constraints–specifically, matching opening and closing tags and ensuring the uniqueness of attributes. Our evaluation focuses on the `xmllint` command line tool that ships with libxml2.

cjson [22] is a popular lightweight JSON parser written in C. We use a JSON input grammar previously mined specifically for cjson [23]. Since cjson does not include a standalone executable, we build a simple command-line tool that reads an input file and passes it to the parser entry point of cjson.

[2]https://www.w3.org/TR/SVG/

All input languages considered in this work are text-based. However, no inherent limitation prevents our approach from being applied to binary input languages. Fandango supports various encodings, bitstrings, endian conversions, and checksum computations through grammar-attached generators[3]. That said, constructing specifications for complex binary formats—such as PNG or JPEG—along with their associated semantic constraints is time-consuming. We leave this for future work, as it lies beyond the scope of this paper.

A. Performance Fuzzing

Fuzzing for performance issues has been explored in multiple prior works [3], [4], [5]. We adopt the *SlowFuzz* approach [3], one of the earliest methods in this space. The basic idea is to guide the fuzzer towards finding inputs that take a long time to execute while keeping the input size small—since longer inputs tend to increase runtime trivially.

To quantify the execution cost of an input, we use the `ExecutionPathLength()` metric, described in Table I. Our instrumentation tracks how often each basic block is executed and aggregates this information into a total path length via `ExecutionPathLength()`.

Consistent with observations in [18], we find that extremely costly inputs can be generated for lunasvg. To maintain experimental practicality, we impose a maximum execution time of ten seconds per input. When an input exceeds this limit, the program is terminated.

Constraint Example 1 (Performance Fuzzing)**.** *In line with prior work [4], [18], we enforce an input length limit of 60 bytes using an input constraint. We model this* performance fuzzing *setup in Fandango using two constraints. First, a* ***hard constraint*** *limiting the input length to 60 bytes*[4]*:*

$$\text{len}(\text{str}(\langle start \rangle)) \leq 60 \tag{6}$$

Second, a ***soft constraint*** *guiding the fuzzer to maximize the execution path length:*

$$\textbf{maximizing}\ \text{DynamicAnalysis}\big(\langle start \rangle\big) \\ .\text{ExecutionPathLength}() \tag{7}$$

Results. We apply this *performance fuzzing* setup to three subjects: lunasvg, graphviz, and libxml2. Results for this

[3]https://fandango-fuzzer.github.io/Binary.html

[4]While these examples attach the execution goal to $\langle start \rangle$, any grammar non-terminal can be used instead.

experiment are shown in the top row of Figure 2. Next, we will detail our findings for each subject.

1) graphviz: When using only execution feedback and a trivial grammar, Fandango struggles to make progress, producing only inputs that have very short execution paths, with practically no improvement over time, likely because Fandango cannot construct valid or even semi-valid graphviz inputs from scratch in this setup. Using only an input specification (no execution feedback), Fandango already generates inputs with longer execution paths. However, improvement stagnates over time as Fandango is unaware of execution path length in this setup. The best results are achieved by combining the input specification with execution feedback. In this setup, inputs run substantially longer, with the best-performing inputs reaching execution path lengths exceeding 20 million basic block hits, which is an improvement of approximately 10x over the specification-only baseline.

2) lunasvg: We observe the same trends as for graphviz. Combining input specification with execution feedback yields the longest execution paths, outperforming both the execution feedback-only and the specification-only configurations.

While performance was the primary focus, our fuzzer also uncovered stability issues in lunasvg. One generated input caused a segmentation fault due to invalid memory write access, which we reported to the developers.[5]. Another input caused the program to hang for over 15 hours; this was also reported as a likely infinite loop.[6]. Both bugs were acknowledged and fixed by the developers.

As discussed, we limited the execution time to ten seconds in the experiments. Hence, inputs generated using execution feedback and the input specification reach a glass ceiling at an execution path length of about twelve billion. Without this limit, even more expensive inputs could be generated.

3) libxml2: Interestingly, libxml2 exhibits different behavior. Although combining an input specification with execution feedback outperforms the specification alone, the execution feedback-only configuration produces inputs with significantly longer execution times. We attribute this to the nature of XML parsers: well-formed inputs (as generated using a specification) are relatively cheap to parse, whereas invalid or semi-valid inputs (as produced by a generic grammar with execution feedback) often trigger expensive error-handling paths, resulting in higher execution costs.

B. Excessive Memory Fuzzing

This domain-specific fuzzing task focuses on discovering inputs that cause programs to allocate excessive amounts of memory. Adopting this domain from FuzzFactory [5], we implement `HeapAllocatedBytes()` (Table I) by instrumenting all `malloc()` and `calloc()` calls in the PUT, and summing the total number of bytes allocated during execution. As in the previous experiment, we enforce a 60-byte input size limit to find short inputs that trigger large memory allocations, possibly indicating a bug.

[5]https://github.com/sammycage/lunasvg/issues/221
[6]https://github.com/sammycage/lunasvg/issues/222

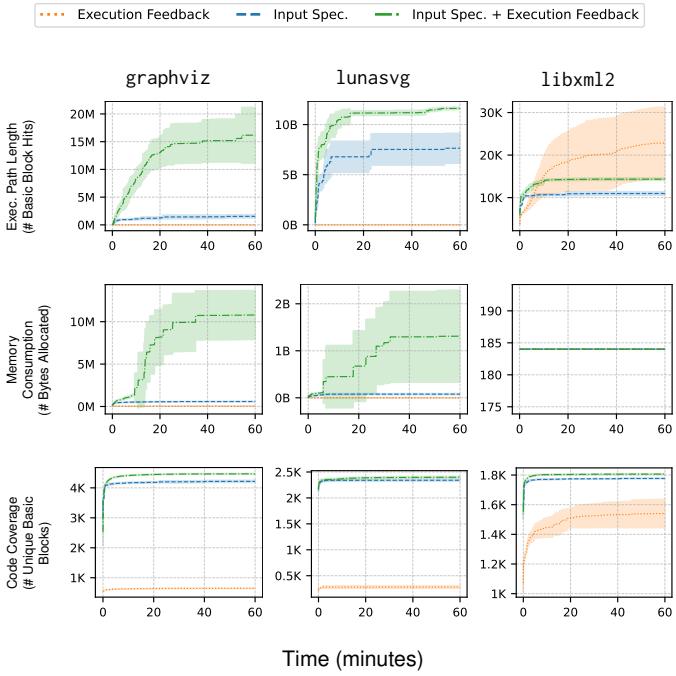

Fig. 2: Results for Section V-A–Section V-C, comparing the performance of (1) execution feedback only, (2) input specification only, and (3) their combination across three domain-specific fuzzing tasks: maximizing execution path length, heap memory consumption, and code coverage. Each experiment ran for one hour and was repeated ten times; bold lines indicate the average, with shaded areas representing the 95% confidence interval.

Constraint Example 2 (Excessive Memory)**.** *The following Fandango constraints model the above setup:*

$$\text{len}(\text{str}(\langle start \rangle)) \leq 60 \tag{8}$$

$$\textbf{maximizing}\ \text{DynamicAnalysis}(\langle start \rangle).\text{HeapAllocatedBytes}() \tag{9}$$

Results. The resulting plots for this experiment are shown in the middle row of Figure 2. Our findings for each of the three subjects are outlined below.

For graphviz and lunasvg, we observe similar trends as in the *performance fuzzing* domain: combining input specifications with execution feedback consistently outperforms using either in isolation. In the case of lunasvg, substantial memory allocations are observed. The input below is one of the costliest, allocating approximately 4 GB of memory in a single call:

```
<svg><path d=" M -279 , 91M 30704 32715Z"/></svg>
```

We investigated this issue and found that the allocation originates from the following code, where `width` and `height` appear to be derived directly from the input:

```
const size_t size = width * height * 4;
plutovg_surface_t* surface =
 malloc(size + sizeof(plutovg_surface_t));
```

In contrast, libxml2 shows constant heap allocations across all configurations, suggesting that it does not allocate heap memory dependent on the input.

C. Maximizing Code Coverage

We introduce this novel fuzzing domain, which aims to identify *individual inputs* that exercise as much code in the PUT as possible. One potential application is finding inputs that simultaneously trigger multiple distinct features, where their interaction—especially in stateful systems—may expose complex, otherwise hard-to-find bugs. We leverage our `CodeCoverage()` metric to implement this domain, which aggregates all executed basic blocks.

Constraint Example 3 (Maximizing Code Coverage)**.** *Our setup requires only one Fandango soft constraint:*

$$\textbf{maximizing}\ \text{DynamicAnalysis}\big(\langle start \rangle\big)\text{.CodeCoverage()} \tag{10}$$

Results. The results for this experiment are shown in the bottom row of Figure 2. Below, we present our findings for each subject in detail.

1) graphviz: Using execution feedback alone results in relatively low basic block coverage. In contrast, both input specification-based setups achieve significantly higher coverage. Notably, combining the input specification with execution feedback yields the highest coverage overall.

2) lunasvg: Again, the two input specification-based setups outperform execution feedback alone. Combining the input specification with execution feedback results in a slight increase in basic block coverage. We attribute this to the input specification being sufficiently succinct and expressive, allowing Fandango alone to achieve broad coverage.

3) libxml2: The combination of input specification with execution feedback again performs best within the one-hour experiment window. Interestingly, execution feedback alone also shows steady progress. Manual inspection of the generated inputs revealed that this setup produces a mix of simple, valid XML inputs and many invalid inputs, contributing to the measured basic block coverage.

D. Directed Greybox Fuzzing

Directed greybox fuzzing focuses on guiding the fuzzer toward specific target locations in the code to find bugs in the vicinity of that code. One application is, for instance, testing recently modified code. Several approaches have been proposed in this area [15], [24], [25], [26], with the seminal work being *Directed Greybox Fuzzing* by Böhme et al. [15]. Our implementation follows a similar approach.

Initially, we perform the following computations once: We extract the call graph of the PUT (consisting of direct calls only) and intraprocedural control-flow graphs for each function. Using breadth-first search, we compute function-level call distances for each pair of functions and basic block-level control-flow distances for each of the two basic blocks in a function. Consistent with [15], these two metrics are combined into a single distance metric by penalizing interprocedural transitions with a constant factor of 10, allowing us to define a distance between any two basic blocks in the program.

During fuzzing, we then use this precomputed information *to calculate the control-flow distance from an input to a target location.* More precisely, for each basic block executed by the input as indicated by the trace, we look up the distance to the target location, returning the minimum value as the result.

We demonstrate directed greybox fuzzing using a small JSON-based experiment. In this setup, a harness wraps the `cjson` parser to parse an input file and then performs a sequence of dependent checks on the parsed JSON structure across three functions: `f01()`, `f234()`, and `f5x()`. The function `f01()` checks that the parsed input is a JSON object containing the keys `"0"` and `"1"`; `f234()` then checks for the presence of keys `"2"`, `"3"`, and `"4"`; finally, `f5x()` looks for keys `"5"` and `"x"`. All these checks are sequentially dependent; for instance, the presence of key `"x"` is only checked for if `"5"` was found. The goal is to reach a specific *target basic block*, which we set to the one executed when `"x"` is present. This setup is illustrated in Figure 3.

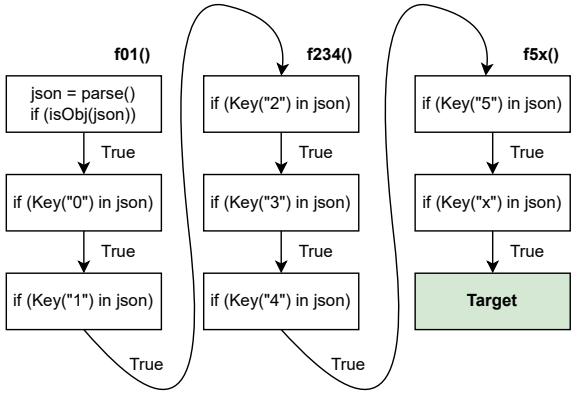

Fig. 3: Schematic control-flow diagram of the `cjson`-based directed greybox fuzzing experiment. All branches must evaluate to `True` to reach the target basic block, which has ID "2" in our Fandango constraint.

The key challenge is that the fuzzer must first generate syntactically valid JSON inputs to pass parsing and then evolve those inputs through mutations to reach the target basic block. To implement this experiment, we configure the fuzzer to minimize the control-flow distance to the *target basic block*, which in this example has the ID `"2"`.

Constraint Example 4 (Directed Greybox Fuzzing)**.** *Again, a single Fandango soft constraint suffices:*

$$\textbf{minimizing}\ \text{DynamicAnalysis}\big(\langle start \rangle\big)\text{.DistanceToBB(“harness.c”, “2”)} \tag{11}$$

We ran this experiment for six hours, comparing fuzzer configurations based on execution feedback, input specifications, and their combination. The results are shown in Figure 4.

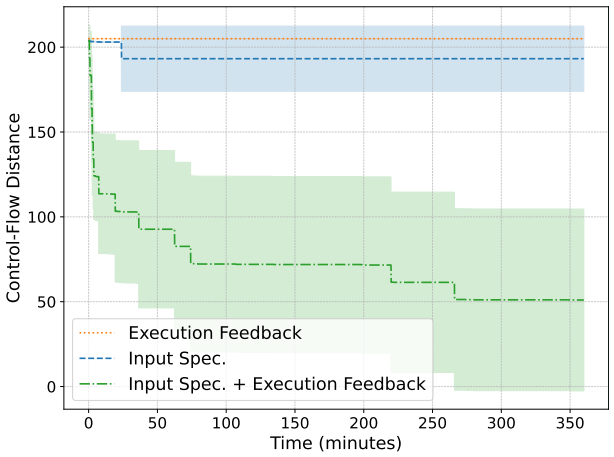

Fig. 4: Result of the directed greybox fuzzing experiment.

Results. The execution feedback-only configuration fails to make progress toward the target. In contrast, the input specification-only configuration makes limited progress and even reaches `f234()` in one of ten runs, as it can generate valid JSON inputs with the first two keys by chance. However, it cannot progress further without execution feedback.

The combined configuration—input specification plus execution feedback—performs best. It makes steady progress and reaches the target basic block in seven out of ten runs within the six-hour experiment, suggesting that FANDANGOGREY effectively combines specification-based input generation with directed fuzzing guided by execution feedback.

One of the target-reaching inputs is shown below. Note that all required keys are present, and the key "X" is uppercase, which works as cjson performs case-insensitive key lookups.

```
{"2":true,"0":"/","5":"",
"4":true,"3":false,"X":"","1":"/"}
```

E. Evolving Populations

The domain-specific fuzzing approaches discussed so far evolve *single inputs*, optimizing them towards desired runtime behaviors. However, during testing, one may not only be interested in a single (failing) test input, but rather in a *population of inputs* that collectively maximize a specific metric such as coverage, which is particularly prevalent in coverage-guided fuzzers like AFL, which also evolve and produce a population of inputs.

By default, the Fandango framework is set to evolve single inputs. However, we can easily set it up to evolve *populations* instead. To this end, we transform the input grammar so that it produces a *population of inputs* instead of just a single input. For example, if the original grammar is defined as (omitting the definition of $\langle input \rangle$ for brevity):

$$\langle start \rangle ::= \langle input \rangle$$

we replace the production rule of the start symbol with:

$$\langle start \rangle ::= \langle input \rangle \{K\}$$

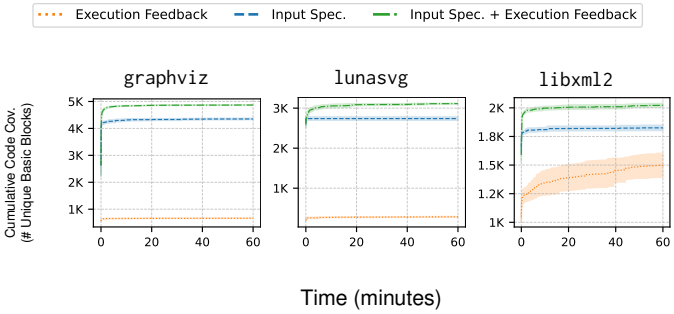

Fig. 5: These results of Section V-E show that coverage-guided fuzzing can be effectively implemented within our framework, and that combining input specifications with execution feedback yields better performance than using either in isolation.

where K is the desired population size, and `{K}` means repetition, as with regular expressions. Thus, Fandango applies mutation and crossover to evolve entire populations of inputs.

During evolution, Fandango does not pass the string derived from $\langle start \rangle$ as the input to the PUT (as it would represent the entire population). Instead, it uses each $\langle input \rangle$ subtree as a separate test input, as enforced by the constraint shown below.

Constraint Example 5 (Evolving Populations)**.** *The following Fandango constraint models this approach:*

$$\textbf{maximizing} \left| \bigcup_{\substack{\langle input \rangle \\ \in \langle start \rangle}} \text{DynamicAnalysis}(\langle input \rangle).\text{CovBBs}() \right| \tag{12}$$

For readability, we present this constraint in mathematical notation. In the actual constraint, we make use of Python built-ins such as `set.union()` and `len()` instead. The $\langle input \rangle$ non-terminals are obtained using Fandango's `children()`[7] function, which returns the direct sub-trees of a derivation tree node. Our primitive `CovBBs()` extracts the set of all basic block IDs covered by a given input from its execution trace, and is also used internally by our `CodeCoverage()` metric.

As in previous experiments, we evaluate this approach by running it on three subjects for one hour each, repeating the experiment ten times. We compare three configurations: using execution feedback alone, using input specifications alone, and using both in combination. For this experiment, we set the population size to $K = 10$.

Results. The experimental results are shown in Figure 5. Consistent with previous experiments, combining input specifications with execution feedback (i.e., coverage guidance) yields the best performance across all subjects, while input specifications alone still outperform execution feedback alone. In addition, for every subject, the cumulative code coverage achieved by the *populations* in this experiment consistently exceeds the single-input code coverage reported in Section V-C, showing the effectiveness of population-based optimization.

[7]https://fandango-fuzzer.github.io/DerivationTree.html#accessing-children

Note that Equation (12) illustrates just *one* goal and just *one* way to combine information from individuals. By modeling populations as individual inputs, FANDANGOGREY can evolve a population towards arbitrary goals regarding input shapes or executions, statistical distributions, or combinations thereof. On top, one could also model the Fandango individuals as *sets of populations*, where each (sub-)population is set to achieve a different goal. All this illustrates the expressiveness and the versatility of FANDANGOGREY.

F. Summary of Results

In summary, our evaluation demonstrates the following:

1) **Specifying fuzzing goals as constraints is very concise.** As shown in constraint examples 1 to 4, FANDANGOGREY requires but *one to two lines of constraints* to express common fuzzing goals, sharply contrasting existing approaches [3], [4], [5], which typically require users to extend the actual implementation.
2) **Constraints can equally refer to inputs and execution.** Constraint examples 1 and 2 demonstrate how to combine constraints related to the input language (say, Eq. (6)) with constraints related to the execution (say, Eq. (7)). Many more such combinations are possible.
3) **Specification-guided (blackbox) fuzzing benefits from execution feedback.** In all our experiments, we found that coverage feedback improves the performance of fuzzing over specification guidance alone.
4) **Coverage-guided (whitebox) fuzzing benefits from input specifications.** In all our experiments, we found that input specifications improve the performance of fuzzing over execution feedback alone. The one exception is libxml2, where *invalid* inputs achieve the highest execution path length; we plan to extend our approach to support such out-of-specification inputs.
5) **Evolving populations.** Our approach allows evolving and producing individuals as well as *populations of inputs,* with goals for each individual or the whole population.
6) **A modular approach.** FANDANGOGREY is the one approach that combines all the above benefits, allowing to specify and combine input languages, blackbox testing goals, whitebox testing goals, and evolution of individuals or populations in a modular and freely adjustable fashion.

VI. THREATS TO VALIDITY

As with all empirical studies, our work is subject to various threats to validity. We outline our mitigation strategies below.

External Validity refers to the extent to which the results can be generalized to other subjects or contexts not included in the study. Even though our evaluation is limited to four programs, we selected a diverse set of real-world programs and input languages, all used in prior fuzzing research, to mitigate this threat. Moreover, while our implementation currently supports a set of nine execution metrics, it can easily be extended, as the underlying evolutionary algorithm is agnostic to the used metric.

Internal Validity refers to whether causal conclusions can be drawn from the results. We mitigated this threat by building our prototype on top of Fandango, which has been used in prior research, and applying it to established fuzzing domains. We also manually verified the correctness of our implementation through unit tests.

Construct Validity refers to the extent to which the chosen metrics accurately measure the intended objectives. We used standard metrics from prior work to mitigate this threat, such as execution path length.

VII. DISCUSSION AND FUTURE WORK

Mining input specifications. As we demonstrated, input specifications can be effectively combined with execution feedback. A key challenge, however, is that input specifications are not always readily available, and their manual construction is time-consuming and error-prone. A potential solution is to automatically mine input specifications from the PUT. Prior work has primarily focused on extracting syntactic input structure, both using blackbox [27], [28], [29] and white-box approaches [30], [31], [32], [23], [33], but largely ignores semantic constraints. As part of our future work, we plan to investigate mining *semantic* input constraints.

Efficient implementation. Our current FANDANGOGREY prototype is based on Fandango, which is implemented in Python. While this facilitates rapid development and flexibility, it introduces a performance overhead compared to highly optimized fuzzers such as AFL++ [17]. Reimplementing Fandango in a compiled language, with efficient interfacing to the target program, could significantly improve performance and scalability.

VIII. CONCLUSION

In this work, we presented FANDANGOGREY, a novel approach for specifying *execution goals in conjunction with input constraints*, enabling rich greybox fuzzing applications where inputs are first shaped according to user-defined constraints and then evolved to optimize execution goals. We implemented a prototype of our approach by extending the Fandango tool with instrumentation-based execution feedback built on LLVM. We provide a comprehensive catalog of domain-specific execution goals that enable users to express their testing objectives in a declarative and user-friendly manner. To demonstrate the practicality and versatility of our approach, we applied it to five domain-specific fuzzing tasks across four real-world C and C++ programs. We believe our approach opens up new possibilities in fuzzing, and we are excited to see how the community will leverage this in future research.

IX. DATA AVAILABILITY

To support future research and enable reproducibility, we release the code, input specifications, and experimental data as open source at:

https://github.com/leonbett/fandangogrey

References

[1] D. Steinhöfel and A. Zeller, "Input invariants," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 583–594. [Online]. Available: https://doi.org/10.1145/3540250.3549139

[2] J. A. Zamudio Amaya, M. Smytzek, and A. Zeller, "Fandango: Evolving language-based testing," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3728915

[3] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2155–2168.

[4] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 254–265.

[5] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[6] D. Steinhöfel and A. Zeller, "Language-based software testing," *Commun. ACM*, vol. 67, no. 4, pp. 80–84, 2024. [Online]. Available: https://doi.org/10.1145/3631520

[7] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 206–215. [Online]. Available: https://doi.org/10.1145/1375581.1375607

[8] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Trans. Software Eng.*, vol. 47, no. 9, pp. 1980–1997, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2941681

[9] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "NAUTILUS: fishing for deep bugs with grammars," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[10] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2018, accessed: 2024-04-09.

[11] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon, "Validity fuzzing and parametric generators for effective random testing," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 266–267. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00107

[12] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 1051–1065. [Online]. Available: https://doi.org/10.1145/3548606.3560602

[13] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: synthesizing structure while fuzzing," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 1985–2002. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[14] T. Dunning, "The t-digest: Efficient estimates of distributions," *Software Impacts*, vol. 7, p. 100049, 2021.

[15] 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.

[16] "Libfuzzer," https://llvm.org/docs/LibFuzzer.html, retrieved 2022-02-01.

[17] 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.

[18] Z. Alsaeed and M. Young, "Finding short slow inputs faster with grammar-based search," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1068–1079.

[19] AT&T Labs Research, "Graphviz: Graph visualization software, version 2.47.0," https://gitlab.com/graphviz/graphviz/-/archive/2.47.0/graphviz-2.47.0.tar.gz, 2021, accessed: 2025-05-01.

[20] S. Ugochukwu, "LunaSVG: SVG rendering library, version 3.2.1," https://github.com/sammycage/lunasvg/releases/tag/v3.2.1, 2025, accessed: 2025-05-01.

[21] D. Veillard and the libxml2 contributors, "libxml2: The XML C parser and toolkit, version 2.9.7," https://gitlab.gnome.org/GNOME/libxml2/-/releases/v2.9.7, 2017, accessed: 2025-05-01.

[22] D. Gamble and contributors, "cJSON: Ultralightweight JSON parser in ANSI C," https://github.com/DaveGamble/cJSON/releases/tag/v1.7.18, 2024, accessed: 2025-05-01.

[23] L. Bettscheider and A. Zeller, "Inferring input grammars from code with symbolic parsing," 2025. [Online]. Available: https://arxiv.org/abs/2503.08486

[24] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 2095–2108. [Online]. Available: https://doi.org/10.1145/3243734.3243849

[25] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1580–1596. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00002

[26] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: A directed greybox fuzzer driven by deviation basic blocks," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2440–2451. [Online]. Available: https://doi.org/10.1145/3510003.3510197

[27] N. Kulkarni, C. Lemieux, and K. Sen, "Learning highly recursive input grammars," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2021, pp. 456–467. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE51524.2021.9678879

[28] M. R. Arefin, S. Shetiya, Z. Wang, and C. Csallner, "Fast deterministic black-box context-free grammar inference," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 117:1–117:12. [Online]. Available: https://doi.org/10.1145/3597503.3639214

[29] F. Li, X. Chen, X. Xiao, X. Sun, C. Chen, S. Wang, and J. Han, "Incremental context-free grammar inference in black box settings," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 1171–1182. [Online]. Available: https://doi.org/10.1145/3691620.3695494

[30] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 720–725. [Online]. Available: https://doi.org/10.1145/2970276.2970321

[31] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 172–183. [Online]. Available: https://doi.org/10.1145/3368089.3409679

[32] L. Bettscheider and A. Zeller, "Look ma, no input samples! Mining input grammars from code with symbolic parsing," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 522–526.

[33] M. Eisele, J. Hägele, C. Huth, and A. Zeller, "GDBMiner: Mining precise input grammars on (almost) any system," *Leibniz Transactions on Embedded Systems*, vol. 10, no. 1, pp. 1–1, 2025.