ACM DIGITAL LIBRARY | Association for Computing Machinery | acm open

Latest updates: https://dl.acm.org/doi/10.1145/3728915

PDF Download
3728915.pdf
28 March 2026
Total Citations: 0
Total Downloads: 838

Published: 22 June 2025
Accepted: 31 March 2025
Received: 31 October 2024

Citation in BibTeX format

RESEARCH-ARTICLE

FANDANGO: Evolving Language-Based Testing

JOSÉ ANTONIO ZAMUDIO AMAYA, CISPA - Helmholtz Center for Information Security, Saarbrucken, Saarland, Germany



Ph.D. Student @CISPA | Research on Systems Security and Fuzz Testing

MARIUS SMYTZEK, CISPA - Helmholtz Center for Information Security, Saarbrucken, Saarland, Germany

ANDREAS ZELLER, CISPA - Helmholtz Center for Information Security, Saarbrucken, Saarland, Germany

Open Access Support provided by:

CISPA - Helmholtz Center for Information Security

FANDANGO: Evolving Language-Based Testing

JOSÉ ANTONIO ZAMUDIO AMAYA, CISPA Helmholtz Center for Information Security, Germany
MARIUS SMYTZEK, CISPA Helmholtz Center for Information Security, Germany
ANDREAS ZELLER, CISPA Helmholtz Center for Information Security, Germany

Language-based fuzzers leverage formal input specifications (*languages*) to generate arbitrarily large and diverse sets of valid inputs for a program under test. Modern language-based test generators combine *grammars* and constraints to satisfy syntactic and semantic input constraints. ISLa, the leading input generator in that space, uses *symbolic constraint solving* to solve input constraints. Using solvers places ISLa among the most precise fuzzers but also makes it slow.

In this paper, we explore *search-based testing* as an alternative to symbolic constraint solving. We employ a genetic algorithm that iteratively generates candidate inputs from an input specification, evaluates them against defined constraints, evolving a population of inputs through syntactically valid mutations and retaining those with superior fitness until the semantic input constraints are met. This evolutionary procedure, analogous to natural genetic evolution, leads to progressively improved inputs that cover both semantics and syntax. This change boosts the efficiency of language-based testing: In our experiments, compared to ISLa, our search-based FANDANGO prototype is *faster by one to three orders of magnitude* without sacrificing precision.

The search-based approach no longer restricts constraints to constraint solvers' (miniature) languages. In FANDANGO, constraints can use the *whole PYTHON language and library.* This expressiveness gives testers unprecedented flexibility in shaping test inputs. It allows them to state arbitrary *goals for test generation:* "Please produce 1,000 valid test inputs where the ⟨*voltage*⟩ field follows a Gaussian distribution but never exceeds 20 mV."

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Empirical software validation**; *Constraint and logic languages*; *Search-based software engineering*; • **Theory of computation →** *Grammars and context-free languages*; *Evolutionary algorithms.*

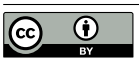
Additional Key Words and Phrases: Language-based testing, fuzzing, test generation

ACM Reference Format:
José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. 2025. FANDANGO: Evolving Language-Based Testing. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA040 (July 2025), 23 pages. https://doi.org/10.1145/3728915

1 Introduction

In the past decade, fuzzers have become the prime tools for detecting software vulnerabilities. Fuzzers randomly generate inputs for the program under test, thus testing its robustness against uncommon inputs. Most of today's fuzzers are coverage-guided. Starting with a population of seed inputs, they mutate and test inputs repeatedly, keeping and further evolving those inputs that get closer to the (uncovered) code of interest.

Authors' Contact Information: José Antonio Zamudio Amaya, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, jose.zamudio@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.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTISSTA040
https://doi.org/10.1145/3728915

```
<PackedFields> ::= <GlobalColorTableFlag> <ColorResolution> \
  <SortFlag> <SizeOfGlobalColorTable>
    <GlobalColorTableFlag> ::= <bit>
    <ColorResolution> ::= <bit>{3}
    <SortFlag> ::= <bit>
    <SizeOfGlobalColorTable> ::= <bit>{3}
<GlobalColorTable> ::= <RGB>*
    <RGB> ::= <R> <G> <B>
      <R> ::= <UBYTE>
      <G> ::= <UBYTE>
      <B> ::= <UBYTE>
```

Fig. 1. A grammar for GIF (excerpt), automatically extracted from a given 010 binary template. The *Kleene star* * denotes zero or more repetitions. The *brackets* {3} denotes 3 exact repetitions.

While this generic approach can be applied to a wide range of programs, it is limited by complex input formats. For example, consider an excerpt of the GIF format shown in Figure 1. Here, the image palette (⟨*GlobalColorTable*⟩) comes as three-byte RGB values. In GIF, the number of entries (colors) in the ⟨*GlobalColorTable*⟩ must equal 2 raised to the power of the value stored in the ⟨*SizeOfGlobalColorTable*⟩ field.

With a format like this, individual mutations to the ⟨*GlobalColorTable*⟩ field without also accordingly adjusting the length of the ⟨*SizeOfGlobalColorTable*⟩ field will result in and invalid GIF.

The recent idea of *language-based testing* [56] addresses this problem by allowing users to supply a *specification* of the input format. This idea as it is is not new: Grammars such as the one in Figure 1 have been used as input producers for decades. But a grammar alone does not suffice to capture such dependencies; indeed, having a grammar producer instantiate strings from Figure 1 alone will hardly ever result in a valid input. The idea of language-based testing is thus to enhance grammars with constraints: predicates over nonterminals that express semantic properties. In the abstract, a constraint such as

$$|\langle \mathit{GlobalColorTable} \rangle| = 2^{\langle \mathit{SizeOfGlobalColorTable} \rangle}$$

where $|x|$ stands for the length of x would thus suffice to specify the relationship between fields.

Besides expressing validity, constraints could also be used to specify *desired properties* of generated inputs. If developers wanted to test *extreme values*, for instance (as mandated by testing standards), they could obtain suitable inputs by specifying predicates such as ⟨*GlobalColorTable*⟩ = 0 or ⟨*SizeOfGlobalColorTable*⟩ = $2^3 - 1$. This ability to *shape* inputs at will, which we term *personalized fuzzing*, goes much beyond the capabilities of common fuzzers, which offer restricted possibilities, let alone a language, to control input generation.

What sounds nice in theory, however, has several deficiencies in practice. ISLa [55], the first fuzzer to combine grammars and constraints, makes use of *symbolic constraint solving* to find solutions for constraints. Giving ISLa a constraint such as `len(<GlobalColorTable>) >= 256`, for instance, will have ISLa produce arbitrary large color tables. However, as ISLa uses symbolic constraint solving, the ISLa constraint language is restricted to functions of the Library of Satisfiability Modulo Theories (SMT-LIB) [3]. Notably, in ISLa, one cannot define functions that could serve as abstractions, meaning that even simple tasks such as converting two bytes into 16-bit unsigned integers have to be explicitly encoded and inlined, and if our image format also contained checksums or hashes (as many real-world image formats do), we cannot express these in ISLa unless we create a specific SMT-LIB theory for them.

```
where len(<GlobalColorTable>) == 2 ** <SizeOfGlobalColorTable>.value()
```

Fig. 2. A FANDANGO constraint for GIF (Figure 1): The lowest three bits of the <PackedFields> field determine the size of the global color table. In FANDANGO, `len(<T>)` returns the number of children of <T>; `.value()` applied to a bit or bit sequence returns its integer value; and `**` is exponentiation in Python.

A second issue is that symbolic constraint solving is powerful but also slow. In a recent study [24], Hasan et al. investigated how well fuzzers produce complex inputs such as the PKCS cryptographic standard for RSA signature schemes. While "ISLa stands out by consistently generating 100% valid inputs for all test subjects" [24] (compared to the best grey-box fuzzer, NAUTILUS [1], with 22%), but also showed "the lowest throughput of 1.30 inputs per second due to its use of an SMT solver during input generation." In contrast, optimized grammar producers have been shown to generate 100,000s of inputs per second [20]—but then without any constraint checking or solving.

In this paper, we address the problems of expressiveness and speed by still relying on language-based testing but using a very different test generation approach. Instead of using symbolic constraint solvers, we use *search-based testing* [37, 38] to find solutions for constraints. Specifically, we use the constraints as *fitness functions,* evolving a population of (given or initially generated) inputs through *syntactically valid* mutations until the semantic constraints are met, giving us three advantages and contributions over the state of the art.

Expressiveness. Using a search-based approach, we can evaluate constraints *concretely* (rather than symbolically). Hence, we can use any general-purpose programming language to express constraints. In FANDANGO, constraints are expressed in PYTHON, tapping into one of the most comprehensive programming ecosystems in the world. For instance, the constraint in Figure 2 suffices to express that the number of entries in the ⟨*GlobalColorTable*⟩ must equal 2 raised to the power of the value stored in the ⟨*SizeOfGlobalColorTable*⟩ field. There are no limitations to function usage, and with PYTHON providing modules for checksums (`crc`), hashes (`hashlib`), and many more, FANDANGO provides unprecedented means for developers and testers to define and shape input formats.
To the best of our knowledge, FANDANGO is the first test generator using a general-purpose programming language to express validity constraints and testing targets.

Speed. All the evolution and fitness assessment of FANDANGO takes place uniquely within the test generator (Figure 3); as with ISLa, the program under test only sees the finally produced valid inputs. However, in our experiments, we found that FANDANGO is faster than ISLa and its symbolic constraint solving *by up to three orders of magnitude,* with the same 100% precision and similar diversity,
Using search-based testing, we improve over the state of the art (symbolic constraint solving) by up to a factor of 1000×.

Abstraction. FANDANGO specifications use a general-purpose programming language for constraints. Yet, they remain *declarative* at their core, giving several advantages over handwritten test generators. In particular, since the syntax is still specified as a grammar, FANDANGO can parse, mutate, and check existing inputs; implementing these capabilities from scratch for each input format would be a pain, neither do testers have to worry about good input generation strategies since these are all implemented once in the fuzzer. The tester has to provide an input specification, though. Once specified, however, it can be reused repeatedly.
A combination of declarative syntax and general-purpose constraints allow for unlimited specification of input formats without sacrificing abstraction and reuse.

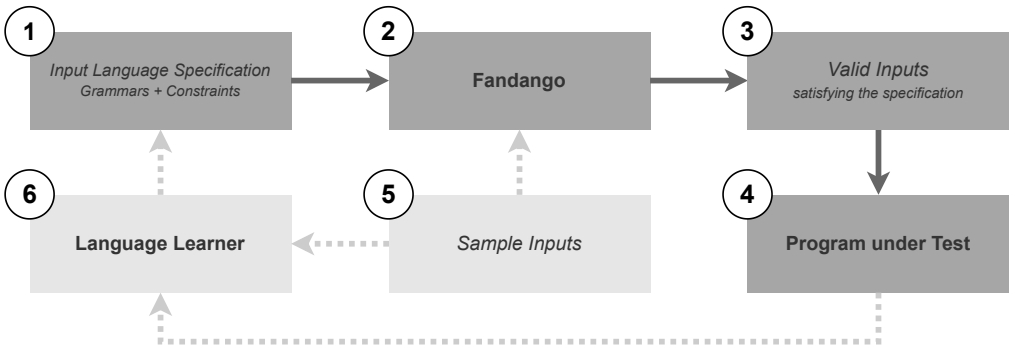


Fig. 3. How FANDANGO works. Given a language specification ① consisting of a grammar (e.g. Figure 1) and constraints (e.g. Figure 1), FANDANGO ② uses *search-based testing* to find solutions for the given specification. These solutions ③ are then used as *test inputs* for the program under test ④. Instead of generating a population from scratch, FANDANGO can also evolve a given population of sample inputs ⑤. If a specification is not available, a language learner ⑥ such as ARVADA [31], MIMID [19], ISLEARN [55], or symbolic parsing [5] can infer input grammars and constraints from given inputs and/or programs.

Having said all this, one should not see FANDANGO as the ultimate test generation tool. If one wants to uncover bugs in parsers, a classic fuzzer that produces plenty of invalid inputs is a better choice. What FANDANGO provides, though, is a means for developers to state formats and goals and a platform to integrate additional tools and features. We are happy to invite developers and researchers to evolve FANDANGO to increase its fitness further.

The remainder of the paper is organized as follows. We discuss the related work in fuzzing and test generation (Section 2). The main contribution comes in Section 3, where we present our approach of evolutionary language-based testing. Section 4 details the algorithmic aspects of our approach. Section 5 discusses implementation details of our FANDANGO prototype. In Section 6, we evaluate FANDANGO, focusing on its expressiveness and speed. After discussing limitations in Section 7, we close with conclusions and future work (Section 8). FANDANGO and all experimental data from this paper are available online (Section 9).

2 Background

Automatically obtaining inputs that would cover the behavior of a given program is a long-standing challenge in computer science. Approaches to test generation at the system level make use of the following ingredients:

Generators. Writing programs that generate inputs for other programs may be as old as programming. *Property-based testing*, introduced with QUICKCHECK [13], automatically produces function inputs to check the function result against a property that should universally hold. ZEST [43] combines property-based testing with coverage feedback (see below) to increase input diversity. AUTOTEST [40] makes use of *contracts* (function pre- and postconditions and data invariants) to produce function inputs and check results. Writing individual generators and/or contracts would allow the individual functions of a program to be thoroughly tested.

Call sequences. In the absence of a test or function specification, function-level generators such as RANDOOP [42] and EVOSUITE [16] make use of the *type system* of the target programming language to generate sequences of function calls that produce the desired objects and invoke methods on these. As they can invoke functions directly, these tools can quickly achieve

high code coverage. However, as types may not capture all functions' preconditions, call sequences may only fail because they violate implicit preconditions. In an experiment on five small-to-medium JAVA programs, RANDOOP reported 181 test failures, all of which were false alarms [21].

Random inputs. Miller's seminal "fuzzing" work [41] showed that one can find severe bugs by simply concatenating random bytes. Simple random fuzzing has become a standard practice to test the robustness of input parsers. However, most inputs will be invalid and not proceed beyond input processing and rejection.

Evolution. Search-based testing [37, 38] starts with a population of valid *seed inputs* which are then randomly mutated; inputs that get closer to the testing goal (typically coverage) are preserved and evolved further. The popular AFL [62] and LIBFUZZER [33] tools implement this concept with high efficiency, using generic mutations at the bit and byte level, and have uncovered thousands of bugs in open-source programs. Central research directions focus on integrating program analysis beyond coverage [49] and strategies to direct the fuzzer towards targets of interests [7, 8]. Regardless of strategy, though, the effectiveness of an evolution-based fuzzer heavily depends on the *diversity* of the seeds: If the seeds miss some feature, evolution-based fuzzers will be challenged to create it through random mutations alone. The field of search-based software testing also has extensive literature on approaches combining heuristics with fuzzing [35, 39]. Many of these studies focus on enhancing the mutation and selection processes to optimize for coverage and structural diversity. However, extending these techniques to address input semantics remains a challenge. This gap in the literature motivates further research into hybrid approaches that combine the strengths of search-based methods with domain-specific semantic analysis.

Constraint solving. In the program code, a *path constraint* collects the programmatic conditions that must be satisfied to reach a particular location. Symbolic test generators use *model checkers* and *constraint solvers* to determine inputs that satisfy path constraints. Seminal representatives include Java Pathfinder [58], KLEE [11] and SAGE [18].

Symbolic analysis has problems with scaling, though. A simple parser for arithmetic expressions already induces far too many constraints for a solver, which are also hard to integrate [6]. Practical approaches thus combine path constraint solving with seed inputs [51], evolution [57], or grammars [17]; or instead make use of lightweight techniques such as gradient descent [12]. Recent work by Liew et al. [34] exemplifies the use of coverage-guided fuzzing to solve floating-point constraints. In their approach, SMT formulas are transformed into programs where a specific location is reachable if—and only if—the input satisfies the underlying constraints. Their method demonstrates that fuzzing can serve as a viable alternative to traditional constraint solvers in certain domains. This observation motivates our work, as FANDANGO extends these ideas by integrating fuzzing with grammar-based test generation to address more general language-based semantic constraints.

Abstract models. Especially in the domain of *cyber-physical systems,* it is common to have *models* that specify system components and their interaction [30]—for instance, using UML [9], MATLAB/SIMULINK [29], or SYSML [25]. These *finite-state models* unlock a wide range of model checking, testing, and verification techniques—but abstract away details about the syntax and semantics of concrete interactions.

Data collections. In 2014, Mariani et al. pointed out the potential of drawing test inputs from existing collections on the Web and suggested integrating these with test generators [36]. *Faker* libraries such as PYTHON's FAKER package [15] provide extensive collections of test data for names, locations, phone numbers, currency codes, and more. As large language models (LLMs) can draw on these collections and other examples, they can synthesize test

data from them. But while having such *common* inputs is valuable to demonstrate basic functionality, systematic testing also requires *uncommon* inputs to cover the entire space of possible inputs—and possible behaviors.

Grammars. If one wants to test a complex program beyond its parser, reliably reach deeper functionality, and cover the entire input space, one needs a *language specification.* Practical use of context-free *grammars* (e.g., Figure 1) for producing test inputs goes back to Burkhardt [10], to be later rediscovered by Hanford [23] and Purdom [48]. Grammars are especially well-suited for complex input languages. The grammar-based LANGFUZZ [27] fuzzer has uncovered over 2,600 bugs in JavaScript interpreters. The PEACH fuzzer [45] comes with hundreds of language specifications ("peach pits") for file formats and network protocols, also using context-free grammars to express input structure. NAUTILUS [1] combines grammar production with coverage feedback from the program. AFLSMART [46] integrates the input-structure component of PEACH with the coverage-feedback component of AFL, thus producing fewer invalid inputs.

Language learners. As specifying and formalizing input languages can be nontrivial, recent *language learners* can infer grammars from given inputs (e.g., ARVADA [31]) and the way these are decomposed by the program under test (e.g., MIMID [19]). *Symbolic parsing* [5] infers precise input grammars from programs *statically*, i.e. not requiring any inputs. The ISLEARN tool learns ISLa *semantic constraints* from sets of valid inputs [55]; similar techniques could be applied to FANDANGO.

Domain-specific generators. Context-free grammars alone are limited in their expressiveness. For instance, when producing program code, it cannot ensure that used identifiers have previously been defined, nor is it well-suited for checksums or length constraints as found in binary inputs. For such complex inputs, *domain-specific generators* such as CSMITH [61] (for C/C++ programs), TLS-Attacker [53] (for TLS interactions), MORPHEUS [59] (for PKCS signature schemes) encompass several person-years of development. Such generators typically are *producers* alone; they do not come with a parser allowing them to reuse or evolve existing inputs.

Constraints. Recent approaches to specifying input languages enhance context-free grammars with *constraints over nonterminals* to express semantic input properties. ISLa [55, 56] uses the SMT-LIB [3] functionality to express input constraints for complex file formats, and the Z3 constraint solver [14] to solve these in conjunction with the grammar. Constraint solvers like Z3 are widely used for solving logical constraints. However, their computational overhead can be significant, especially when dealing with considerable constraints or intricate input specifications. In a recent comparison of fuzzers for complex inputs [24], ISLa stood out "by consistently generating 100% valid inputs for all test subjects", but also with the "lowest throughput of 1.30 inputs per second due to its use of an SMT solver during input generation." While ISLa has been demonstrated to be sufficiently expressive for several file formats [55], the underlying SMT-LIB functionality can be surprisingly limited—for example, the SMT-LIB `str.to_int` function, converting strings to integers, accepts positive integers only [3].

In summary, we conclude that

(1) for complex input languages, *expressive specification languages* continue to be a necessity;
(2) modern *language learners* can significantly reduce the effort for specifying grammars and constraints;
(3) short of implementing domain-specific fuzzers, *combinations of grammars and constraints* are the most promising way to consistently achieve high precision; and
(4) relying on *symbolic constraint solvers* slows input generation considerably.

It is these observations that have led us to the development of FANDANGO—achieving *high precision* with *high efficiency* and *high expressiveness.*

3 Evolutionary Language-Based Testing

Evolutionary language-based testing, as we have named it, is an approach that combines principles from evolutionary algorithms [4] with language-based software testing [56] to explore the input space of systems systematically. This approach leverages the syntactical structure of inputs defined by grammars and applies evolutionary operators to generate syntactically valid and semantically meaningful inputs.

3.1 Syntax

To produce syntactically valid inputs in the first place, *context-free grammars* are employed alongside constraints over grammar rules. Formally, a context-free grammar G is a set of production rules R, where:

(1) R is a finite set of production rules, each of the form $A \to \alpha$, where:
   - $A \in V$ is a nonterminal symbol and
   - $\alpha \in (V \cup \Sigma)^*$ is a string of terminals and/or nonterminals;
(2) V is a finite set of nonterminal symbols (variables);
(3) Σ is a finite set of terminal symbols, disjoint from V; and
(4) G must have a start symbol S, that indicates the first rule R to expand.

To better illustrate this formal definition, we will use the grammar in Figure 1 to guide our explanations. In our example grammar, R refers to the rules we can find in the grammar. For instance, the rule `<PackedFields> ::= <GlobalColorTableFlag> <ColorResolution> <SortFlag> <SizeOfGlobalColorTable>`, expresses that every `<PackedFields>` element is formed by the concat of `<GlobalColorTableFlag> <ColorResolution> <SortFlag> <SizeOfGlobalColorTable>`. We will find a rule of the form $A \to \alpha$ in R that expresses how each *nonterminal symbol* V is derived. Following this example, we will continue expanding rules until reaching a rule of the form $A \to \alpha$, where α cannot be expanded further, the so-called *terminal symbols*. At the end of our example grammar, we can find the rule `<bit> ::= "0" | "1"`. This means, every nonterminal symbol `<bit>` is either `0` or `1`, chosen randomly. The nonterminal `<SizeOfGlobalColorTable>` will be a concatenation of three `<bit>`, (`0`s and `1`s), and in this way, we can trace back every nonterminal to the *root*, ending the expansion process and returning a valid string that is syntactically valid towards our grammar. Following this procedure, we can formally specify the details of any language specification, easily producing syntactically valid complex inputs.

3.2 Semantics

Semantic constraints and predicates can be challenging to encode in solvers, thus limited to a small, inexpressive language to define constraints. We have designed evolutionary language-based testing to be expressive and not limited by a subset of specific predicates. For this reason, constraints are encoded as arbitrary PYTHON code, specifying semantic predicates over nonterminal elements in the grammar. Formally, the constraint language supports the following constructs:

nonterminals. The construct `<T>` refers to the string nonterminal T from the grammar.

Universal quantifiers. The construct `forall V in <T>: P` is true if and only if *all* occurrences V of the nonterminal $\langle T \rangle$ satisfy P.[1]

Existential quantifiers. The construct `exists V in <T>: P` is true if at least *one* occurrence V of the nonterminal $\langle T \rangle$ satisfies predicate P.

[1]Future Fandango versions will replace these by Python quantifiers (`all()`, `any()`) over sets of nonterminals.

Selection operators. The construct <T>.<T'> refer to every direct child node T' of a node T in the derivation tree. The form <T>..<T'> refers to all descendant nodes T' of a node T.

PYTHON features. Besides the above, all of PYTHON features, including arithmetic, comparisons, logic, as well as standard PYTHON functions and user-defined functions, can be used.

We will use the constraints from Figure 2 to better illustrate this formal definition. Here the length of <GlobalColorTable> must be equal to two to the power of the integer value of <SizeOfGlobalColorTable>. Since the constraint refers to nonterminals from the grammar, we can efficiently compute if this constraint is satisfied dynamically.

3.3 Initialization

After parsing the grammar and encoding the constraints, we can produce an initial population of *derivation trees* based on G. Formally, a *derivation tree* T, also known as a parse tree, is a rooted, ordered tree that represents the syntactic structure of a string according to a given grammar $G = (V, \Sigma, R, S)$. It demonstrates how a string in a language is generated by the production rules R. We can define the structure and validity of a derivation tree as follows:

(1) **Structure of a Derivation Tree**:
  (a) **Node Types and Labels**:
    - Each node in the tree is labeled with a symbol from $V \cup \Sigma$, where V is the set of nonterminal symbols, and Σ is the set of terminal symbols.
    - The root node is labeled with the start symbol S.
  (b) **Internal Nodes and Production Rules**:
    - Internal nodes are labeled with nonterminal symbols from V.
    - If an internal node is labeled with a nonterminal $A \in V$ and has children labeled $X_1, X_2, \ldots, X_n$, there exists a production rule $A \rightarrow X_1 X_2 \ldots X_n$ in R.
  (c) **Leaf Nodes**:
    - Leaf nodes are labeled with terminal symbols from Σ or the empty string ϵ.

(2) **Validity of a Derivation Tree with Respect to a Grammar**:
  (a) **Root Node Requirement:**
    - The root node r must be labeled with the start symbol S of the grammar.
  (b) **Leaf Node Requirement**:
    - If a node n is a leaf (i.e., $\text{children}(n) = \emptyset$), then its label must be a terminal symbol from Σ or the empty string ϵ.
  (c) **Internal Node Requirement and Production Rules**:
    - If a node n is an internal node (i.e., $\text{children}(n) \neq \emptyset$), then its label must be a nonterminal symbol from V.
    - There must exist a production rule in R that matches the label of n with the labels of its children. Specifically, if n has children labeled $c_1, c_2, \ldots, c_k$, then the production rule $\text{label}(n) \rightarrow \text{label}(c_1)\,\text{label}(c_2)\ldots\text{label}(c_k)$ must exist in R.
  (d) **Yield of the Tree**:
    - The yield of the tree (the concatenation of the labels of the leaf nodes from left to right, excluding ϵ) forms a string $w \in \Sigma^*$ that is generated by the grammar G.

Using *derivation trees* instead of simple *text strings* to represent inputs, we can trace each element back to its corresponding production in the grammar. This tracing is crucial for ensuring maximum control over the grammar elements and identifying which parts of the input contribute to constraint violations. As an example, let us use the grammar in Figure 1 to produce a derivation tree. In Figure 4 we can see a partial example, where each nonterminal V tracks down its children expansions.

Every nonterminal eventually leads to a terminal symbol, so the combination of terminal symbols in the tree leaves represents the input generated by the grammar. For example, if we wanted to specify a constraint over every `<bit>` nonterminal derived from `<SizeOfGlobalColorTable>`, we could use a `forall` constraint selecting `<SizeOfGlobalColorTable>.<bit>`. With this technique, we have a method to specify constraints over any combination of grammar elements, allowing us to specify constraints over combination of elements and its children.

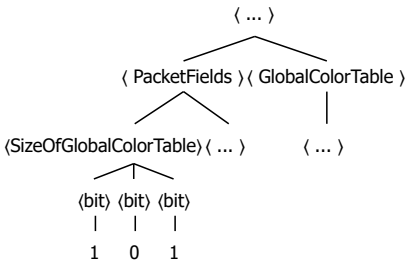


Fig. 4. Partial derivation tree produced by the grammar in Figure 1.

3.4 Population Evaluation

After generating the initial population of derivation trees, each individual is evaluated based on how well it satisfies the specified constraints. The evaluation process involves traversing the derivation tree and checking each constraint against the relevant nodes and sub-trees.

A fitness function $f : T \to \mathbb{R}$ assigns a fitness score to each derivation tree T, quantifying its suitability. Formally, the fitness function is defined as:

$$f(T) = \sum_{c \in C} w_c \cdot s_c(T)$$

where:

- w_c is the weight assigned to constraint c, reflecting its importance.
- $s_c(T) \in [0, 1]$ is the satisfaction score of tree T with respect to constraint c.

Calculating fitness scores guides the selection process toward more promising solutions. For our running example based on Figure 1 and Figure 2, this process should be executed as follows:

Extracting values. Traverse T to extract the values specified in the constraints. In our example, we extract the strings representing `<GlobalColorTable>` and `<SizeOfGlobalColorTable>`.

Data transformation. Convert the extracted values into the required types or formats for constraint evaluation (e.g., converting strings to integers). In our example, we convert the string of `<SizeOfGlobalColorTable>` into an integer value using the `.value()` function.

Constraint checking. Evaluate each constraint $c \in C$ by applying it to the transformed data, determining whether the constraint is fully satisfied or violated. In our example, we determine the actual length of `<GlobalColorTable>` and compare it against the value of `<SizeOfGlobalColorTable>` executing `eval(2 ** <SizeOfGlobalColorTable> == <GlobalColorTable>)` [2].

Fitness score assignment. For each constraint:

- If the constraint is fully satisfied, assign a maximum score $s_c(T) = 1$.
- If the constraint is violated, assign a score $s_c(T) = 0$.
- For some constraints, we can assign a score $s_c(T)$ reflecting the degree of compliance[3], typically in the range $[0, 1]$.

[2] PYTHON `eval()` function returns always a string. Automatic type conversions are integrated to work with arbitrary types.
[3] The compliance varies based on the type of constraint. For instance, the compliance is based on the distance between the expected value and the input value for numerical constraints.

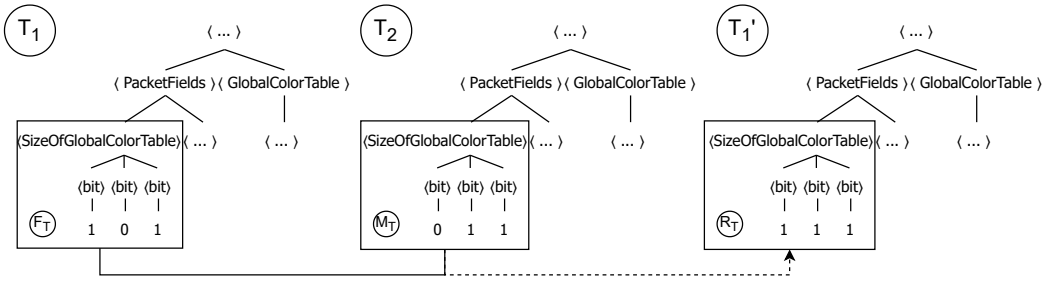


Fig. 5. Crossover of two derivation trees (T_1) and (T_2) derived from the grammar in Figure 1. After identifying a failing tree (F_T), a matching sub-tree (M_T) is selected and integrated, resulting in a new tree (T_1').

3.5 Crossover

Crossover is the primary operator for recombining two *derivation trees* to produce offspring. Each pair of parent derivation trees, T_1 and T_2, have a set probability to be crossed. This probability is guided by each individual's fitness, meaning that the fittest individual will produce more offspring, creating the next generation of derivation trees. If a combination of parents is selected, they will produce offspring, recombining parts of each parent T_1 and T_2 into two new offspring trees T_1' and T_2'. The crossover operation identifies nodes associated with constraint violations in T_1 and T_2. We identify which sub-trees in the derivation tree fail to satisfy a constraint for each parent. We have named this specific type of sub-trees as *failing trees* F_t. We randomly select a F_t in T_1, and we find the corresponding matching sub-tree in T_2. The crossover is produced by swapping the selected sub-trees between the two parents to create two new children, T_1' and T_2', added to the next generation. By performing this operation with matching sub-derivation trees, we ensure that the offspring remain syntactically valid and potentially combine beneficial traits from both parents. This recombination could lead to new inputs that are both syntactically and semantically valid. Figure 5 shows an example crossover between two parents derived from the grammar in Figure 1.

3.6 Mutation

Mutation introduces random variations into the population, aiding in exploring the search space and preventing premature convergence. Mutation operations are designed to modify derivation trees while maintaining syntactical validity. For a given derivation tree T, the mutation process is performed by selecting a *mutation point*, which is a node n associated with a constraint violation in T. After identifying a faulty node n, we generate a new sub-tree T_{new} from the same nonterminal symbol in the grammar, replacing the sub-tree rooted at n with T_{new} to produce the mutated tree T' (Figure 6). By regenerating the sub-tree from the same nonterminal, the mutation operation preserves the syntactical correctness of the derivation tree but explores different possibilities within the grammar, increasing the coverage of the grammar features.

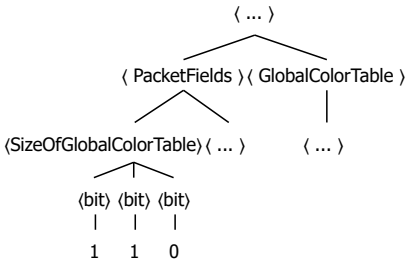


Fig. 6. Mutation example in the generated derivation tree (T_1'). The last $\langle bit \rangle$ has been mutated using the grammar in Figure 1.

Algorithm 1 Evolutionary Algorithm F($G = (S, R), C, N, G_{\max}, R_C, R_M$)

```
P_0 ← GenerateInitialPopulation(G, N)
δ ← 0
while δ < G_max do
    for all T ∈ P do
        fitness(T) ← EvaluateFitness(T, C)
    end for
    if TerminationConditions(P) then
        break
    end if
    while |P_{δ+1}| < N do
        P_parents ← ParentSelection(P)
        P_{δ+1} ← ApplyCrossover(P_parents, R_C)
    end while
    for all T ∈ P_{δ+1} do
        ApplyMutation(T, R_M)
    end for
    P ← P_{δ+1}
end while
return P
```

3.7 Algorithm Workflow

The evolutionary process iterates through evaluation, selection, crossover, and mutation until the population satisfies all the constraints, or it stagnates, and no further improvement is made after several generations. Once terminated, the algorithm outputs the best-performing individuals as test inputs for the system under test.

Given a grammar G, a set of constraints C, a population size N, a number of max generations $G_{\max}$, a crossover probability R_C and a mutation probability R_M, the evolutionary algorithm is summarized in Algorithm 1.

4 Algorithm Details

The evolutionary language-based testing approach discussed in Section 3 is implemented accordingly in FANDANGO. The implementation encompasses parsing the grammar and constraints, generating the initial population, evaluating individuals against constraints, and applying evolutionary operators to evolve the population toward satisfying all constraints. An ANTLR parser [44] has been implemented to parse the grammars and constraints, using version 4.13.2. The entire codebase is implemented in pure PYTHON, focusing on a modular design that can be extended further. By providing the grammar and the constraints, FANDANGO works automatically. However, it can be customized by modifying its parameters:

- `population_size`: specifies the number of individuals.
- `max_generations`: specifies the number of generations until termination.
- `elitism_rate`: specifies the % of individuals that shall be passed onto the next generation without modification.
- `tournament_size`: specifies the % of individuals that shall be used for each tournament.
- `crossover_rate`: specifies the crossover probability for each pair of individuals.
- `mutation_rate`: specifies each individual's probability of suffering a mutation.

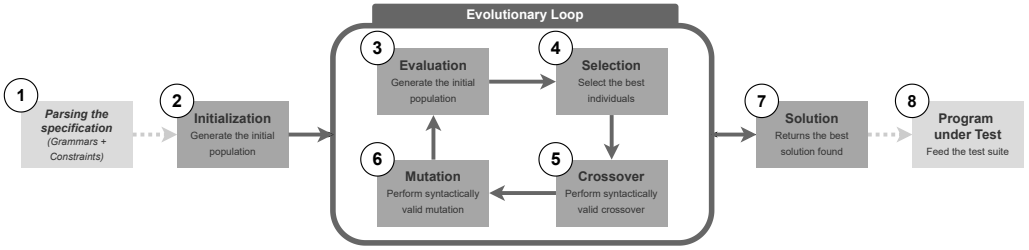


Fig. 7. The FANDANGO evolutionary approach. Given a language specification, FANDANGO ① encodes the constraints. Fuzzing from the grammar, FANDANGO ② produces an initial set of inputs, which will be evolved ② through the evolutionary loop over a number of generations. The inputs will be evaluated ③ based on how well they satisfy each constraint, and the best will be selected ④ to produce offspring ⑤. The new generation of inputs will be mutated ⑥ in order to further differentiate them from the previous generation. Then, the loop repeats, until the constraints are satisfied. After finding a valid set of inputs ⑦, FANDANGO uses them as *test inputs* for the program under test ⑧.

Constraints are parsed and transformed into *evaluable* functions that can be applied to derivation trees. With the grammar, FANDANGO produces an initial population of size `population_size` by randomly expanding the grammar G. After producing the initial population, each individual is evaluated based on how well it satisfies each constraint. This evaluation is the most expensive process in the framework. Still, optimizations, such as persisting an in-memory cache with partially evaluated trees, are computed and used to speed up the process.

In FANDANGO, the selection of pairs of derivation trees to perform crossover is performed based on fitness scores to choose individuals for reproduction. Specifically, FANDANGO uses the *elite selection* [47] and the *tournament selection* [60] methods. Although this process could lead to randomly selecting a subset of poor individuals, ensuring the solutions do not fall into a local optimum is essential, balancing selection pressure and maintaining diversity. The selection starts with the best `elitism_rate%` of the population is passed onto the next generation. After this, FANDANGO randomly selects a subset of `tournament_size` individuals from the population. The individual with the highest fitness in the tournament is selected as a parent, and this process is repeated by selecting `population_size` - (`population_size` * `elitism_rate%`) parents, which will be potentially crossed.

Once the pairs of parents have been selected, the crossover recombines them to produce offspring. Each pair of T_1 and T_2 has a probability of `crossover_rate%` to be crossed. After the crossover, each individual has a `mutation_rate%` probability of being further mutated.

FANDANGO repeats this process for up to `max_generations` generations Extra termination conditions, such as early termination (every individual in our population satisfies every constraint) or stagnation (no significant improvement in fitness scores over `max_generations * 0.2` generations), are also implemented. Over successive generations, individuals evolve toward satisfying the constraints, resulting in syntactically and semantically valid inputs. We can see the program workflow in Figure 7.

5 Implementation Details

User Interface. FANDANGO comes as a *command-line tool,* with dedicated commands for producing and parsing inputs, such as `fandango fuzz -f spec.fan` or `fandango parse -f spec.fan -i input.txt`. Output and algorithm settings can be controlled using command-line options.

Integration. To integrate FANDANGO into existing testing frameworks, the command-line interface is the easiest way. A Python API is also available.

String encodings. By default, FANDANGO produces strings in UTF-8 Unicode encoding. Prefixing a string with `b` (say, `<separator> ::= b'\xff'`) makes it *binary*, preventing UTF-8 interpretation, and allowing for arbitrary byte sequences.

Bits. FANDANGO allows terminal symbols to be *bits* (`0` or `1`). During production, bits are composed into bytes with the most significant bit coming first; during parsing, bytes are decomposed into bits in the same fashion.

Repetitions. FANDANGO allows the common shorthands `+`, `*`, `{n}`, and `{n,m}` to specify repetitions of grammar symbols, such as `<byte> ::= <bit>{8}`.

Regular expressions. When a string is prefixed with `r`, it is interpreted as a regular expression (a *regex;* say, `<char> ::= r"(.|\n)"`). During fuzzing, regexes are efficiently instantiated into strings. During parsing, inputs are efficiently matched against regexes; this allows for the classical separation of lexing (with regexes) and parsing (with grammars).

Libraries. FANDANGO includes a standard library with common definitions, such as `<char>` or `<digit>`. An `include` function allows importing and specializing other FANDANGO specs.

Python code. Anything in a FANDANGO spec that is not a grammar or a constraint is interpreted as Python code, thus allowing for defining functions, constants, or importing modules.

6 Evaluation

To evaluate FANDANGO, we compare our approach against ISLa [55], the state-of-the-art language-based fuzzer that can satisfy syntactic and semantic constraints. ISLa generates inputs that adhere to formal grammars, ensuring syntactic validity while satisfying semantic constraints by leveraging a constraint solver. To the best of our knowledge, ISLa is the only tool with these capabilities. In contrast to ISLa, FANDANGO employs a heuristic method to satisfy constraints, providing a distinct mechanism to address the same challenges. Given ISLa's unique position as the only fuzzer capable of achieving both syntactic and semantic validity, it is our natural choice for benchmarking.

To ensure a fair comparison, we will use *the same benchmark suite* that ISLa employs in its evaluation [55]—that is, CSV, XML, TAR, ScriptSizeC (a simplified subset of C similar to TinyC), and reST (reStructuredText) formats. First, we will re-evaluate ISLa on its original suite to establish a consistent baseline. Then, we will adapt the grammars and constraints from the ISLa benchmark suite to our language, allowing for a direct comparison under identical testing conditions.

6.1 Research Questions

To evaluate FANDANGO's effectiveness and compare it to ISLa, we have formulated research questions targeting performance, diversity, conciseness, and expressiveness.

RQ1: How *fast* is FANDANGO compared to ISLa? This question is our key point, pitching FANDANGO's evolutionary test generation vs. ISLa's constraint solver.

RQ2: How *diverse* are the inputs produced by both tools? Higher performance must not come with a loss of diversity, so we compare the output diversity of ISLa and FANDANGO.

RQ3: How *concise* are the respective outputs? This question is more informative, as short and long inputs have value, but we are curious to find differences here.

RQ4: How *expressive* is FANDANGO? We explore some constraint types that only FANDANGO can handle.

RQ5: Can one define complex inputs in FANDANGO? We explore two complex data formats in GIF and SVG case studies.

6.2 Evaluation Setup and Preliminary Results

All tests were conducted on a machine with the following specifications:

- CPU: Apple M2 Max 12-core CPU with 8 performance cores and 4 efficiency cores. [4].
- RAM: 32GB unified memory.

To reproduce ISLa results, we followed the reproduction guidelines provided in the submitted replication package for ISLa and initially attempted to use the latest version of ISLa. However, due to instability issues that caused the latest version to crash, we reverted to the version referenced in the original guidelines to ensure consistent and reliable comparisons. To reproduce the results of our tool, FANDANGO, please refer to the instructions available at the FANDANGO official repository.

Both tools run for an hour, producing as many inputs as possible for each subject. After the execution is complete, both tools report the inputs produced, alongside measurements on the number of valid inputs produced per minute, the grammar coverage [26] based on k-path with k=4, and the mean and median length of the input. Then, a subject-specific parser is invoked to check how many generated inputs are valid. Both tools use the same parsers to ensure a fair comparison. The final results are shown in Table 1.

Table 1. Results obtained after running each tool on every subject for exactly an hour. For each of our five subjects, we report the precision of the fuzzer, which represents the percentage of inputs produced by the fuzzer that are accepted by the subject parser. The number of inputs accepted by the subject parser, the coverage of the grammar features up to depth four, and the mean and median length of the valid inputs are shown. Each subject details the parser and version used.

Subject	Tool	#Inputs per second	Precision	Coverage 4-path	Speedup	Mean length	Median length
CSV 1.0	FANDANGO	**2,792.39**	99.74%	100.00%	**1,355**×	10.26	11
(PYTHON csv)	ISLa	2.06	100.00%	100.00%		1,212.33	837
reST 0.21.2	FANDANGO	**1,195.34**	99.27%	100.00%	**146**×	15.25	14
(PYTHON docutils)	ISLa	8.18	100.00%	95.13%		31.96	32
ScriptSizeC 0.9.28rc	FANDANGO	**200.75**	99.51%	96.00%	**29**×	24.81	24
(PYTHON tccbox)	ISLa	6.85	100.00%	55.67%		29.05	30
TAR 3.5.3	FANDANGO	**13.33**	100.00%	74.00%	**48**×	2,048.00	2,048
(bsdtar)	ISLa	0.28	100.00%	87.42%		4,025.03	4,096
XML 1.3.0	FANDANGO	**433.70**	99.67%	100%	**183**×	21.60	23
(PYTHON xml)	ISLa	2.38	100.00%	90.50%		45.10	49

6.3 RQ1: How *fast* is FANDANGO compared to ISLa?

Consider the leftmost column "#Inputs per second" in Table 1. Across all subjects, FANDANGO consistently outperformed ISLa, achieving significant speedups. For example, with the CSV subject, FANDANGO generated inputs at a rate of 2792.39 inputs per second compared to ISLa's 2.06 inputs per second, resulting in a speedup of **1,355**×. Similarly, for reST, FANDANGO achieved a speedup of **146**×, and for ScriptSizeC, it achieved a **29**× speedup. FANDANGO can also produce valid complex inputs with *cyclic constraints* such as TAR files, at a much faster speed than ISLa. The "Speedup" column shows the respective speedup.

On the ISLa benchmark suite, FANDANGO is faster than ISLa by a factor between 29× (ScriptSizeC) and 1,355× (CSV), producing hundreds to thousands of valid inputs per second.

[4] FANDANGO and ISLa do not include parallelization options, and they are not designed to be run using parallel processing. For this reason, the evaluation is run in a single core.

The column "Precision" shows how many of the generated inputs are also accepted by the respective processing programs; we see that this is also 100% or almost a 100%, indicating the correctness of tools and specs. However, it is important to notice that the heuristic nature of FANDANGO could result in test suites with less than 100% valid inputs, indicating that we might need to increase the number of generations.

6.4 RQ2: How *diverse* are the inputs produced by both tools?

The k-path measure captures the variety of syntactic paths covered within the derivation trees of a grammar, providing a measure of the syntactic diversity achieved [26]. The value k stands for the length of the paths; "full 4-path coverage" thus means that in the derivation trees, the inputs cover all possible subpaths of length 4.

The "Coverage" column in Table 1 shows the 4-path grammar coverage achieved by both tools. FANDANGO achieved 100% 4-path coverage in almost all subjects, matching or surpassing ISLa's coverage. Notably, for ScriptSizeC, FANDANGO achieved 100% 4-path coverage, substantially higher than ISLa's 55.67%, demonstrating FANDANGO's ability to explore a broader range of input features. For TAR, FANDANGO achieved 74% 4-path coverage, 13.42% less than ISLa. This might be due to deriving the grammar from the specification, resulting in partially two different grammars, with more features. It should be noted that the coverage is normalized according to time; in the same amount of time, FANDANGO covers more grammar features than ISLa. It may be that each input from FANDANGO covers fewer features (thus being more concise), but the high input generation rate compensates for this.

On average, FANDANGO's outputs cover more grammar features and combinations than ISLa's.

6.5 RQ3: How *concise* are the respective outputs?

Regarding the conciseness of inputs, the columns "Mean length" and "Median Length" in Table 1 show that FANDANGO consistently produces smaller inputs in every subject, which, by itself, is neither feature nor flaw. Smaller and longer inputs can be fundamental in testing, as both hold intrinsic value and in several contexts [50], minimal test cases can be even more effective than more prolonged test cases. However, it is essential to notice that extra constraints ensuring input length manipulation can be added easily into every subject's constraints with FANDANGO.

It's also worth mentioning that for the CSV subject, FANDANGO's inputs often consist of a shorter line or columns. In contrast, ISLa 's longer average input lengths might result from attempts to maximize coverage within each input. However, this subject does not require constraints regarding the input size to be valid.

FANDANGO's outputs are more concise than ISLa's.

6.6 RQ4: How *expressive* is FANDANGO?

FANDANGO is designed to handle a broader range of constraint types than existing tools, including ISLa. This expressiveness enables FANDANGO to generate data with complex dependencies beyond plain syntactic or simple semantic constraints. To illustrate FANDANGO's expressiveness, we showcase three examples that ISLa cannot handle using the grammar in Figure 8.

External data collections. FANDANGO can draw values from external data collections, allowing for more contextualized and realistic data generation. For example, we use the `Faker` library to generate realistic names by drawing from a pool of existing data (see `fake_name` function in Figure 9). In contrast, ISLa lacks the flexibility to integrate external libraries, limiting its ability to access real-world data sources for contextual data generation.

```
<start> ::= <data_record>
<data_record> ::= <str> ' = ' <hash> ' = ' <int>
<str> ::= <char>+
<hash> ::= <hexdigit>+
<int> ::= <digit>+
```

Fig. 8. A simple grammar `simple.fan` that produces a name, a hash, and an integer. The nonterminals <char>, <hexdigit>, and <digit> are predefined in the FANDANGO standard library.

```
import random
import hashlib
from faker import Faker

def fake_name():
    fake = Faker()
    return str(fake.fake_name())

def valid_hash(string):
    return str(hashlib.sha256(str(string).encode('utf-8')).hexdigest());

def gaussian_integer():
    value = random.gauss(100, 50)
    return str(round(value))

include('simple.fan')
where str(<str>) == valid_hash(<string>)
where str(<hash>) == valid_hash(<string>)
where str(<int>) == gaussian_integer()
```

Fig. 9. Specializing `simple.fan` (Figure 8). The FANDANGO constraints (introduced by a `where` keyword) state that <name> has to be drawn from a pool of existing names, <hash> has to be the SHA-256 hash of the generated name, and <int> needs to follow a Gaussian distribution across the entire population.

Generating valid hashes. FANDANGO allows the generated hash to be computed dynamically based on the value of the generated name. As shown in Figure 9, we define a custom function `valid_hash` using PYTHON's `hashlib` library to compute the SHA-256 hash of the generated name string. ISLa does not support dynamic hash generation, as it relies on solvers that cannot handle such procedural computations.

Sampling from statistical distributions. FANDANGO allows the generation of values that follow desired distributions. In this example, we generate integers that follow a Gaussian (normal) distribution centered around a mean of 100 with a standard deviation of 50. This distribution is defined using the custom `gaussian_integer` function shown in Figure 9. ISLa cannot enforce this type of constraint, as it lacks support for probabilistic constraints and distribution-based sampling within its constraint-solving framework.

By leveraging PYTHON's full expressiveness for constraints, FANDANGO can generate complex, contextually accurate, and semantically valid data that reflects real-world conditions more closely than traditional grammar-based fuzzers. This expressiveness highlights FANDANGO's versatility and capability to handle a wide range of constraints beyond what ISLa and other tools can achieve.

FANDANGO can draw on a large ecosystem of functions unavailable for symbolic analysis.

```
<ellipse> ::= '<ellipse' (<ws> <ellipse_attribute>)* ('/>' | '>' (<desc> | \
<title> | <metadata> | <animate> | <set> | <animateMotion> | \
<animateColor> | <animateTransform>)* '</ellipse>')
<ellipse_attribute> ::= 'id=' <ellipse_id_value>
    | 'transform=' <ellipse_transform_value>
    | 'ry=' <ellipse_ry_value>   # required
    | 'rx=' <ellipse_rx_value>   # required
    | 'cy=' <ellipse_cy_value>
    | 'cx=' <ellipse_cx_value>
    | (... 104 more attributes ...)
where ('ry=' in <ellipse>.descendant_values() and
       'rx=' in <ellipse>.descendant_values())
```

Fig. 10. A FANDANGO spec for SVG (excerpt), automatically extracted from the SVG document type definition (DTD). The method `.descendant_values()` returns the list of values (as strings) of all descendants of a node.

6.7 RQ5: Can one define complex formats in FANDANGO?

We close our evaluation with two case studies, exploring the generation of complex data for the popular GIF and SVG image formats. Both formats are challenging to generate due to their complex structure and dependencies between elements.

GIF (Graphics Interchange Format) encodes images as pixels in a binary format. Rather than writing a FANDANGO GIF spec from scratch, we wrote a *converter script* that would take an existing *binary template* for the 010 Editor [52] and convert it into a FANDANGO spec. The result of applying this script to `gif.bt` is shown in Figure 1. The original `gif.bt` has 160 lines of code; the FANDANGO spec has 114 lines of code.

Our converter script can only partially handle the C-like *code* included in 010 templates which computes lengths and other values. For these, we added constraints as shown in Figure 2 manually to the FANDANGO spec. We also had all GIFs produced contain only one pixel, such that we would not have to implement the LZW compression for pixel data. With these extra constraints, FANDANGO would generate valid 100% GIF files; we found that some transparent pixels would render differently in the Chrome, Safari, and Firefox browsers.

SVG (Scalable Vector Graphics) encodes images as vectors using a text-based XML format. Again, rather than writing an SVG spec from scratch, we wrote a script that would take an XML *document template definition* file (a DTD, such as the DTD file for SVG) and convert it into a FANDANGO spec. The result of applying this script to `svg.dtd` is shown in Figure 10. The original DTD for SVG has 4,509 lines of code; the FANDANGO spec, expanding all macros, has 8,000 lines of code.

One of the challenges in generating XML files, including SVG, is that some XML nodes require specific attributes and elements to be present, but not necessarily at a specific position. With FANDANGO, we can easily express that required children be present (for ellipses, the `rx` and `ry` attributes, for instance), without specifying their order.

The DTD does not specify the valid values for individual elements; adding these to the spec (say, `<rx_value> ::= <Length_datatype>` and `<Length_datatype> ::= '"' <int> '"' | "'" <int> "'" | <int>`) added another 250 lines of spec.

With all this, we could go and generate SVG files. Rather than producing entirely random SVG files, we concentrated on *extreme values* for all numerical attributes, which we could easily produce through the respective constraints. For instance, a constraint such as (`<Length_datatype> == "'1000000'"`) would produce SVG elements with large dimensions

and thus stress-test SVG parsers. With its "vast test suite that includes around 1,600 tests", the `resvg` SVG renderer [2] passes more SVG tests than any other SVG renderer. However, when confronted with `<svg>` image sizes of 1,000,000 pixels, `resvg` 0.44.0 crashes.[5]

FANDANGO can express complex binary and structured formats for practical testing purposes.

6.8 Threats to Validity

While FANDANGO demonstrates promising results in generating syntactically and semantically valid inputs across diverse scenarios, certain limitations may impact the reliability of our findings.

Construct validity. FANDANGO relies on fitting constraints and evaluating them based on the specified constraints to guide the evolution of inputs. While this approach generally works well, it can struggle with constraints that require indirect feedback or involve abstract properties. For instance, if a complex constraint combines multiple interdependent conditions, the fitness score may not provide accurate signals, leading to premature convergence on near-valid solutions that do not fully satisfy the constraints. This limitation is inherent to fitness-guided approaches designed to approximate constraint satisfaction rather than ensure it.

Internal validity. Since genetic algorithms are stochastic, their outcomes may vary between runs due to random mutations, crossovers, and other probabilistic elements. Moreover, variations in the choice of parameters (e.g., population size, mutation rate, crossover rate) can influence the solution quality and convergence speed. Our evaluation suite includes multiple well-known scenarios where *grammar-based fuzzing* is practically applied to account for this inherent variability, allowing us to examine FANDANGO's performance across a diverse range of conditions.

External validity. To evaluate FANDANGO's applicability and performance in real-world use cases, we have designed the evaluation suite to cover diverse scenarios where grammar-based fuzzing is commonly used. We compared FANDANGO against ISLa [55], the state-of-the-art tool for generating syntactically and semantically valid inputs, providing a benchmark for speed and accuracy in constraint satisfaction.

7 Limitations

While *genetic algorithms* offer efficient ways to generate inputs that aim to satisfy a set of constraints, several inherent limitations should be clearly stated. These limitations stem from the *heuristic nature* of genetic algorithms and the *specific problem structure* in constraint satisfaction tasks. Below, we outline the most significant limitations of our approach:

Non-monotonic constraints. One of the key limitations of our approach is its struggle with constraints that provide no clear guidance on how to improve toward satisfaction. Non-monotonic constraints lack the property that more minor changes in the input will lead to incremental improvements toward satisfying the constraint. In these cases, the algorithm randomly explores the solution space without any meaningful guidance from the fitness function. Although this randomness allows the algorithm to generate many potential solutions quickly, the lack of directional feedback significantly reduces the chances of finding a valid solution. In contrast, exact algorithms, which systematically explore the solution space, might be slower but more reliable in such cases. Approaches such as the ones discussed in [22, 32, 54] discuss different ways to handle with such constraints.

[5] All bugs have been reported to the respective developers.

Guarantees of optimality. Genetic algorithms are inherently stochastic and heuristic. Therefore, our approach does not provide any guarantee that the best solution will be found. While these algorithms are practical in exploring large search spaces and can converge on a valid or high-quality solution, they are not guaranteed to find the globally optimal solution or even a valid one, depending on the complexity of the problem and constraints. For applications that require finding the exact best solution, our approach may not suffice, and an exact algorithm would be necessary.

Unsatisfiable constraints. Another significant limitation is the inability to detect when constraints are unsatisfiable. The genetic algorithm will continue generating and evaluating solutions without indicating whether a valid solution exists. Instead, it will continue indefinitely or until the predefined generation limit is reached, potentially wasting computational resources. In contrast, exact algorithms often incorporate mechanisms that allow them to determine if no solution exists within certain boundaries, thus saving computational effort by stopping early.

Problem suitability for heuristic approaches. Not all problems are suitable for heuristic algorithms like genetic algorithms. Specific problems, especially those with complex interdependencies between constraints, are poorly suited to heuristic approaches. For example, problems with highly constrained, narrow solution spaces (e.g., combinatorial optimization problems with strict feasibility requirements) may lead the algorithm to repeatedly produce infeasible solutions. The random and iterative nature of the algorithm may struggle to make meaningful progress in such cases. Exact methods are often better suited to handling these problems. They can systematically explore the solution space and guarantee the discovery of valid solutions when they exist.

Lack of parallelization for high-dimensional input spaces. Implementing parallelization and optimization techniques, such as distributed processing or parallel constraint evaluation, could enhance FANDANGO 's efficiency and scalability for large-scale input generation tasks. Without such optimizations, FANDANGO 's performance may be limited when generating extensive or computationally intensive inputs.

These limitations suggest that while FANDANGO can be highly effective for the evaluation subjects we have used, it involves some trade-offs that may lead to poor solutions depending on the nature of the problem. For applications requiring guarantees of *correctness*, *optimality*, or *dissatisfaction detection*, other methods may be more appropriate.

8 Conclusion and Future Work

To solve input constraints and produce valid complex inputs, *search-based testing* is an efficient alternative to symbolic constraint solving. Our FANDANGO input generator is faster than the state of the art by an order of magnitude or more without sacrificing precision.

Search-based testing allows constraints to be expressible in general-purpose programming languages by no longer being limited to symbolically analyzable functions. In FANDANGO, constraints can use the entire PYTHON ecosystem, giving testers unprecedented flexibility in specifying input formats and goals for test generation.

We anticipate several future usages for tools like FANDANGO. Besides generic tasks such as documentation and bug fixing, our future work will focus on the following topics:

Coverage guidance. While FANDANGO is conceived as a black-box input generator, one can also use it in a white-box manner—by integrating the program under test into the fitness function. A constraint (or fitness function) like `coverage(program_under_test) >= 0.8`, where `coverage` would return the coverage achieved by the population, would turn FANDANGO

into a coverage-driven test generator, evolving the population using syntax-based mutations towards the stated coverage goal. We are currently experimenting with instrumenting C code to obtain measures such as branch or condition distance to leverage these in FANDANGO.

Execution features. Besides coverage, one can think of other desirable execution features to be achieved in testing—for instance, reaching a particular location in the code or some internal variable getting a particular value. We are experimenting with primitives that allow us to specify and formulate such goals.

Protocols. Grammars are not limited to inputs alone. *I/O grammars* [28] integrate inputs and outputs in a single grammar, allowing testers to specify the syntax of interaction between multiple agents. Adding constraints to I/O grammars makes it possible to express the semantics of interactions ("If you send X, I'll send you Y with the following properties"); the expressiveness of PYTHON could make these full protocol specifications.

Compilation. Converting grammars into compilable and executable code can yield extremely fast producers [20]. By compiling constraints to a low-level language as well, the speed-up might be orders of magnitude.

Symbolic Reasoning. Symbolic analysis could help to solve complex constraints or to detect infeasible constraints. We expect interesting synergies between symbolic reasoning and evolutionary algorithms.

Synergies with Traditional Fuzzers. The seeds generated by FANDANGO are semantically valid and diverse, making them ideal candidates to serve as initial inputs for traditional coverage-guided fuzzers. By leveraging FANDANGO 's capability to produce high-quality, constraint-compliant seeds, traditional fuzzers can further mutate these inputs to explore deeper execution paths and uncover additional bugs.

Security Applications. FANDANGO addresses two fundamental challenges in language-based testing: *slow input production* and *complex constraint specification.* These advancements are particularly valuable for detection of vulnerabilities. With FANDANGO, we are now able to rigorously test complex formats and extend our approach to other security-critical domains, thereby broadening the scope and impact of language-based fuzzing.

Acknowledgments. Leon Bettscheider, Marcel Böhme, Addison Crump, Alessandra Gorla, Alexander Liggesmeyer, Tim Scheckenbach, the participants and organizers of the Río Cuarto Summer School, and the anonymous reviewers provided valuable feedback and suggestions. Thanks a lot!

This research was funded by the European Union (ERC S3, 101093186). Views and opinions expressed are however those of the authors 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.

9 Data Availability

We are fully committed to open science, and the FANDANGO official repository is open source. FANDANGO and all experimental data from this paper are available online at

https://github.com/fandango-fuzzer/fandango

The repository includes information on how to install FANDANGO and details of the evaluation process used in this paper. The included tutorial and reference contains over 160 pages of in-depth documentation, including guidance on running FANDANGO and example use cases. Ongoing projects and future work can be found in the Issues section.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars.. In *NDSS*.

[2] Resvg Authors. 2025. resvg—An SVG rendering library. https://github.com/linebender/resvg.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). https://smt-lib.org

[4] Thomas Bartz-Beielstein, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. 2014. Evolutionary algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4, 3 (2014), 178–195.

[5] Leon Bettscheider and Andreas Zeller. 2024. 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* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 522–526. https://doi.org/10.1145/3663529.3663790

[6] Nikolaj Bjørner. 2022. Z3 fails to terminate despite set timeout. https://github.com/Z3Prover/z3/issues/5891#issuecomment-1063110825.

[7] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 2, Article 7 (June 2018), 52 pages. https://doi.org/10.1145/3210309

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[9] Grady Booch. 2005. *The Unified Modeling Language user guide*. Addison Wesley.

[10] W. H. Burkhardt. 1967. Generating test programs from syntax. *Computing* 2, 1 (01 Mar 1967), 53–73. https://doi.org/10.1007/BF02235512

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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* (San Diego, California). USENIX Association, USA, 209–224. https://dl.acm.org/doi/10.5555/1855741.1855756

[12] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 711–725. https://doi.org/10.1109/SP.2018.00046

[13] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[15] Daniele Faraglia. [n. d.]. The Python Faker module. https://faker.readthedocs.io/

[16] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Szeged, Hungary). ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[17] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Tucson, AZ, USA). ACM, 206–215. https://doi.org/10.1145/1375581.1375607

[18] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (January 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[19] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 172–183. https://doi.org/10.1145/3368089.3409679

[20] Rahul Gopinath and Andreas Zeller. 2019. Building Fast Fuzzers. arXiv:1911.07707 [cs.SE] https://arxiv.org/abs/1911.07707

[21] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. EXSYST: Search-based GUI testing. In *2012 34th International Conference on Software Engineering (ICSE)*. 1423–1426. https://doi.org/10.1109/ICSE.2012.6227232

[22] Bernd Gruner, Clemens-Alexander Brust, and Andreas Zeller. 2025. Finding Information Leaks with Information Flow Fuzzing. *ACM Trans. Softw. Eng. Methodol.* (January 2025). https://doi.org/10.1145/3711902 Just Accepted.

[23] K. V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Systems Journal* 9, 4 (December 1970), 242–257. https://doi.org/10.1147/sj.94.0242

[24] S Mahmudul Hasan, Polina Kozyreva, and Endadul Hoque. 2024. FuzzEval: Assessing Fuzzers on Generating Context-Sensitive Inputs. arXiv:2409.12331 [cs.CR] https://arxiv.org/abs/2409.12331

[25] Matthew Hause. 2006. The SYSML modelling language. In *Fifteenth European Systems Engineering Conference*, Vol. 9. 1–12. https://www.omgsysml.org/The_SysML_Modelling_Language.pdf
[26] Nikolas Havrikov and Andreas Zeller. 2020. Systematically covering input structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 189–199. https://doi.org/10.1109/ASE.2019.00027
[27] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security Symposium*. USENIX Association, Bellevue, WA, 38–38. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler
[28] Barry Jones, Mark Harman, and Sebastian Danicic. 1999. *Automated Construction of Input and Output Grammars*. Technical Report. University of North London. https://www.researchgate.net/publication/2448384_Automated_Construction_of_Input_and_Output_Grammars
[29] Steven T Karris. 2006. *Introduction to Simulink with engineering applications*. Orchard Publications.
[30] Stuart Kent. 2002. Model driven engineering. In *International conference on integrated formal methods*. Springer, 286–298. https://dl.acm.org/doi/10.5555/647983.743552
[31] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2022. Learning highly recursive input grammars. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 456–467. https://doi.org/10.1109/ASE51524.2021.9678879
[32] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1 (2018), 1–13.
[33] LibFuzzer [n. d.]. LibFuzzer. https://llvm.org/docs/LibFuzzer.html. Retrieved 2022-02-01.
[34] Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. 2019. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 521–532. https://doi.org/10.1145/3338906.3338921
[35] Jan Malburg and Gordon Fraser. 2011. Combining search-based and constraint-based testing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 436–439. https://doi.org/10.1109/ASE.2011.6100092
[36] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2014. Link: exploiting the web of data to generate test inputs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/2610384.2610397
[37] Phil McMinn. 2004. Search-Based Software Test Data Generation: A Survey. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156.
[38] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.
[39] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 153–163. https://doi.org/10.1109/ICSTW.2011.100
[40] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. 2009. Programs That Test Themselves. *Computer* 42, 9 (2009), 46–55. https://doi.org/10.1109/MC.2009.296
[41] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (December 1990), 32–44. https://doi.org/10.1145/96267.96279
[42] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902
[43] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576
[44] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.
[45] Peach fuzzer [n. d.]. Peach fuzzer, community edition. https://gitlab.com/peachtech/peach-fuzzer-community. Retrieved 2022-02-12.
[46] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering (TSE)* 47, 9 (2019), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681
[47] Dipesh Pradhan, Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. 2017. CBGA-ES: A cluster-based genetic algorithm with elitist selection for supporting multi-objective test optimization. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 367–378.

[48] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (1972), 366–375. https://doi.org/10.1007/BF01932308

[49] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Network and Distributed System Security Symposium (NDSS)*. http://dx.doi.org/10.14722/ndss.2017.23404

[50] PG Sapna and Arunkumar Balakrishnan. 2015. An approach for generating minimal test cases for regression testing. *Procedia computer science* 47 (2015), 188–196.

[51] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Lisbon, Portugal). ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[52] Sweetscape Software. 2025. 010 Editor—Binary Template Repository. https://www.sweetscape.com/010editor/repository/templates/.

[53] Juraj Somorovsky. 2016. Systematic Fuzzing and Testing of TLS Libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1492–1504. https://doi.org/10.1145/2976749.2978411 Code available at https://github.com/tls-attacker/TLS-Attacker.

[54] Philip Sperl and Konstantin Böttinger. 2019. Side-Channel Aware Fuzzing. In *Computer Security – ESORICS 2019*, Kazue Sako, Steve Schneider, and Peter Y. A. Ryan (Eds.). Springer International Publishing, Cham, 259–278.

[55] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3540250.3549139

[56] Dominic Steinhöfel and Andreas Zeller. 2024. Language-Based Software Testing. *Commun. ACM* 67, 4 (March 2024), 80–84. https://doi.org/10.1145/3631520

[57] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Network and Distributed System Security Symposium (NDSS)*, Vol. 16. 1–16. https://sites.cs.ucsb.edu/~vigna/publications/2016_NDSS_Driller.pdf

[58] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (Boston, Massachusetts, USA) *(ISSTA '04)*. Association for Computing Machinery, New York, NY, USA, 97–107. https://doi.org/10.1145/1007512.1007526

[59] Moosa Yahyazadeh, Sze Yiu Chau, Li Li, Man Hong Hue, Joyanta Debnath, Sheung Chiu Ip, Chun Ngai Li, Endadul Hoque, and Omar Chowdhury. 2021. Morpheus: Bringing the (PKCS) one to meet the oracle. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2474–2496.

[60] Jiaping Yang and Chee Kiong Soh. 1997. Structural optimization by genetic algorithms with tournament selection. *Journal of computing in civil engineering* 11, 3 (1997), 195–200.

[61] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Jose, California, USA) *(PLDI '11)*. ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[62] Michal Załewski. 2018. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2024-04-09.

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