Constraint-Driven Fuzzing at Scale with FANDANGO

José Antonio Zamudio Amaya, Marius Smytzek,
Alexander Liggesmeyer, Valentin Huber, and Andreas Zeller
CISPA Helmholtz Center for Information Security
Germany
{jose.zamudio, marius.smytzek, alexander.liggesmeyer, valentin.huber, zeller}@cispa.de

***Abstract*—By combining grammars with semantic constraints, language-based testing enables the generation of complex, valid test inputs. We present FANDANGO, a test generator that replaces symbolic solving with genetic algorithms, allowing constraints to be written directly in Python rather than in a limited function-solver library. Compared to the state-of-the-art, this approach delivers input generation speeds one to three orders of magnitude higher while maintaining near-perfect validity.**

FANDANGO is easy to use and supports a wide range of input formats. This paper emphasizes the public release: packaging, command-line, and Python APIs, reproducible examples, and early adoption in teaching and industry.

A screencast is available at: https://youtu.be/7W9CaHEC1bM

***Index Terms*—Language-based testing, fuzzing, test generation**

I. Introduction

Automated testing depends on the quality and diversity of test inputs. Complex programs expect inputs that are not merely well-formed according to a structural specification (a grammar, a data format, a protocol), but that also satisfy semantic value-level constraints: a network packet's length field must match its payload; a CSV row for a financial application must contain only non-negative balances. Finding inputs that simultaneously satisfy all such requirements is a fundamental challenge in automated test generation.

Grammar-based testing [1] addresses the structural side by encoding input formats as context-free grammars and generating syntactically valid, diverse inputs by derivation. Figure 1 shows a simple grammar. *Language-based testing* [2] extends this to *semantic constraints* (predicates over the constituents of the derived string). Figure 2 shows different constraints that we can impose on grammar elements.

In this paper, we demonstrate FANDANGO [3], an open-source, Python-based language-based test generator that replaces symbolic solving with a *genetic algorithm*. Whereas our earlier paper focused on the technique and its performance, this submission focuses on the released tool and its practical reuse. A FANDANGO specification combines a context-free grammar with constraints written in plain Python, in contrast to SMT formulas. Starting from a population of syntactically valid inputs, FANDANGO evolves that population via grammar-aware crossover and mutation, evaluating each individual's fitness against the constraints, and selecting the fittest to seed the next generation, until all constraints are satisfied.

Because constraints are ordinary Python, testers can harness statistical distributions, external validators, or custom oracles to articulate goals such as *"produce 1,000 valid inputs where the int field follows a Gaussian distribution centered at 100 with a deviation of 50."*

This work is partially 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.

```
<start> ::= <data_record> ;
<data_record> ::= <str> '=' <hash> '=' <int>;
<str> ::= <char>* ;
<hash> ::= <hex_digit>* ;
<int> ::= <digit>* ;
<char> ::= "A" | "B" | "C" | "D" | "E" | ... ;
<hex_digit> ::= "0" | "1" | "2" | "3" | ... ;
<digit> ::= "0" | "1" | "2" | "3" | ... ;
```

Fig. 1. A simple grammar that produces a name, a hash, and an integer

```
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))

where str(<str>) == fake_name();
where str(<hash>) == valid_hash(<str>);
where str(<int>) == gaussian_integer();
```

Fig. 2. FANDANGO constraints for the grammar in Figure 1. The name must be drawn from a pool of existing names, the hash must be the SHA-256 hash of the generated name, and the integer must follow a Gaussian distribution across the entire population.

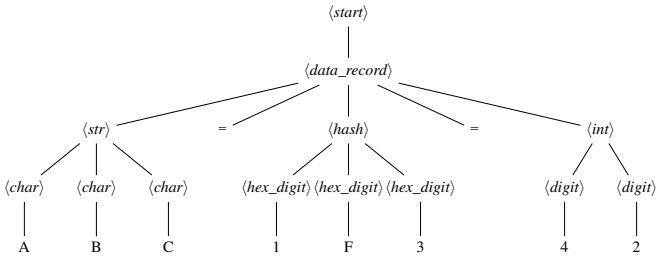


Fig. 3. Derivation tree for the record `ABC=1F3=42`

II. The FANDANGO Fuzzer

Evolutionary Language-based Testing combines principles from evolutionary algorithms [4] with language-based software testing [2] to systematically explore the input space of a system under test. It leverages the structure of inputs defined by grammars and applies evolutionary operators to generate syntactically valid and semantically meaningful inputs.

FANDANGO relies on two core ingredients: a *context-free grammar* (Figure 1) that describes the structure of the input language, and a set of *constraints* (Figure 2) that capture semantic requirements. The grammar recursively expands nonterminal symbols until reaching terminal characters (Figure 3), guaranteeing syntactic validity by construction. Constraints go beyond structure, capturing properties that the grammar alone cannot enforce. FANDANGO encodes them as arbitrary PYTHON `where` predicates. Our distinction between *value-shaping* and *quantifier* constraints is explanatory rather than syntactic: the former constrain one field or subtree, while the latter apply `any()` or `all()` to collections of matched nodes. Optimization goals such as minimization and maximization are also supported.

Internally, FANDANGO represents inputs as *derivation trees* rather than plain strings. Figure 3 shows an example of such a structure. A derivation tree mirrors the grammar's expansion steps: the root corresponds to the start symbol, each internal node to a grammar rule application, and the leaves to the actual characters of the input. Reading the leaves from left to right reconstructs the input string. This tree structure is crucial because it allows FANDANGO to pinpoint exactly which part of an input is responsible for a constraint violation.

A. Workflow

FANDANGO evolves a population of inputs through evaluation, selection, crossover, and mutation, until all constraints are satisfied or no further progress is made over consecutive generations. The workflow is summarized in Figure 4. Starting from a random population expanded from the grammar, each input is scored by how well it satisfies the constraints: 1 if fully met, 0 if violated, and partial values in between reflecting proximity to the target. High-scoring individuals are selected as parents and combined through *crossover*, which identifies the *failing sub-trees* responsible for constraint violations and swaps matching sub-trees between parents. Because only structurally compatible pieces are exchanged, offspring are guaranteed to remain syntactically valid. The *mutations* then replace a failing sub-tree with a freshly generated one from the same grammar rule, introducing variation and preventing the population from converging prematurely.

B. Installation

FANDANGO is distributed as a Python package on PyPI, installable with a single command:

```
pip install fandango-fuzzer
```

The command will install FANDANGO and all its dependencies. The tool can be used via a command-line interface or imported as a library in Python scripts. Example specifications and usage instructions are provided in the documentation[1].

C. Using FANDANGO

To illustrate how FANDANGO's constraint types compose in practice, we use the running example grammar from Figure 1, which produces records of the form `str=hash=int`. This specification could represent a simple database record format: a name, its cryptographic digest, and a numeric identifier. Without constraints, FANDANGO freely expands each nonterminal, producing syntactically valid but semantically arbitrary records. By layering constraints of increasing expressiveness, we progressively narrow the output space toward meaningful test inputs, as shown in Figure 2.

1) Regular Expressions: First, let us shape the *lexical structure* of individual tokens. Instead of enumerating every letter as a grammar alternative, we replace the `<char>` rule with a compact regular expression:

```
<char> ::= r'[A-Za-z]'
```

This expression restricts `<str>` to alphabetic characters only.

2) Python Generator Functions: We can use *generator functions*, arbitrary PYTHON code attached directly to a grammar rule via the `:=` syntax. We define two functions: `valid_hash`, which computes the SHA-256 digest of `<str>`, and `gaussian_integer`, which samples from $\mathcal{N}(100, 50)$:

```
<hash> ::= <hex_digit>* := valid_hash(<str>)
<int> ::= <digit>+ := gaussian_integer()

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

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

Once attached, `<hash>` is no longer randomly expanded; it is always the cryptographic digest of whatever `<str>` was generated, and `<int>` follows a realistic statistical distribution. This example illustrates how generator functions can encode *semantic dependencies* between fields that a grammar alone cannot express.

[1]The 353 pages of documentation on language-based testing with FANDANGO can be accessed at: https://fandango-fuzzer.github.io/

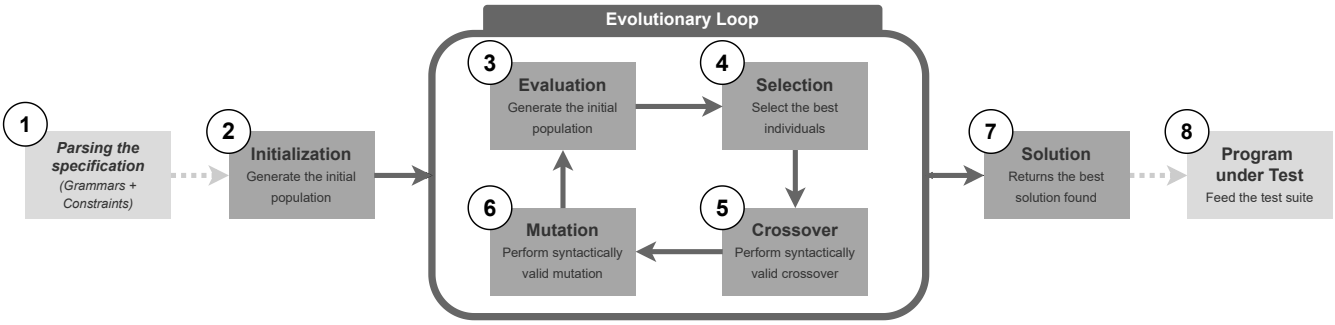


Fig. 4. The FANDANGO evolutionary approach. Given a language specification, FANDANGO encodes the constraints (1) and fuzzes the grammar to produce an initial input population (2), which is then evolved over successive generations. Each generation, inputs are evaluated against the constraints (3), the fittest are selected (4) to produce offspring (5), and the new generation is mutated (6) to maintain diversity. This loop repeats until all constraints are satisfied, at which point the valid inputs (7) are delivered as test inputs to the program under test (8).

3) Value-Shaping Constraints: We can impose *arithmetic and structural bounds* on one field or local subtree with the `where` keyword:

```
where 3 <= len(str(<str>)) <= 10
where int(<int>) != 113
```

The first bounds `<str>` to 3–10 characters; the second excludes 113 from `<int>`.

4) Quantifier Constraints: We can also constrain several matched nodes at once. The `*<char>` notation collects all `<char>` nodes in the current input as a PYTHON iterable. With `any()` or `all()`, this yields existential or universal quantification; `not(any(...))` is just composition:

```
where any(str(c) in 'AEIOUaeiou' for c in *<char>)
```

This constraint requires at least one vowel in `<str>`; unlike the value-shaping constraints above, it ranges over all expanded `<char>` nodes.

5) Optimization Goals: FANDANGO also supports optimization goals, which are separate from boolean constraints. Minimization and maximization guide the search toward smaller or larger values rather than merely accepting or rejecting inputs.

III. Validation and Experience

The original evaluation of FANDANGO was conducted in our previous work [3]. Here, we summarize those results and focus on what is new in this tool paper: availability, usability, classroom experience, and use in industry.

A. Experimental Setup

To evaluate FANDANGO, we compare our approach against ISLA [5]. ISLA generates inputs that adhere to formal grammars, ensuring syntactic validity, and leverages a constraint solver to enforce semantic constraints. In contrast, FANDANGO employs a heuristic, evolutionary method to satisfy constraints, providing a distinct mechanism to address the same challenges.

To ensure a fair comparison, we use the same benchmark suite as ISLA's original evaluation [5]: CSV, XML, TAR, ScriptSizeC (a simplified subset of C), and reST formats. First, we re-evaluate ISLA on its original suite to establish a consistent baseline. Then, we adapt the grammars and constraints from the ISLA benchmark suite to FANDANGO's language, allowing for a direct comparison under identical testing conditions. We ran each tool for an hour on each subject, measuring the number of generated inputs per second and the percentage of valid inputs according to the specification.

TABLE I
EVALUATION RESULTS ACROSS FIVE SUBJECT PROGRAMS.

	Approach	inputs/s	Validity (%)	Speedup
CSV 1.0	ISLA	2.06	100.00	1,355×
	FANDANGO	2,792.39	99.74	
reST 0.21.2	ISLA	8.18	100.00	146×
	FANDANGO	1,195.34	99.27	
ScriptSizeC 0.9.28rc	ISLA	6.85	100.00	29×
	FANDANGO	200.75	99.51	
TAR 3.5.3	ISLA	0.28	100.00	48×
	FANDANGO	13.33	100.00	
XML 1.3.0	ISLA	2.38	100.0 0	183×
	FANDANGO	433.70	99.67	

B. Results

Table I summarizes the results of our evaluation, comparing FANDANGO with ISLA for multiple subjects. FANDANGO achieves a significant speedup of 1 to 3 orders of magnitude while maintaining an almost as high validity as ISLA, demonstrating that heuristic optimization can effectively replace symbolic solving in language-based testing.

C. Application in Industry Settings

Recent work stemming from Volkswagen demonstrates the value of language-based test generation in industry. AUTOSPEC [6] uses large language models to synthesize formal protocol specifications from natural-language RFCs and uses FANDANGO as the downstream test generator, illustrating reuse beyond stand-alone fuzzing and showing that FANDANGO fits naturally into toolchains with automatically synthesized specifications, not only manually written ones. More broadly, this suggests that FANDANGO can serve as an execution backend for emerging specification-synthesis workflows in industrial practice. This kind of reuse is especially valuable when generated specifications still need expressive semantic checks during test generation.

D. Experience and Lessons Learned

FANDANGO has been used in university courses on software testing[2] and summer schools[3], where hundreds of students applied it to diverse formats and protocols. Students could move quickly from tutorial grammars to their own formats, making FANDANGO practical for hands-on exercises and small course projects. This quick adaptation also made it suitable for short teaching units in which students had to understand both syntax and semantic constraints within limited class time. In practice, this let courses emphasize input modeling and testing goals rather than the mechanics of a separate constraint language. Qualitatively, feedback centered on direct Python constraints, one-file specifications, and reuse of the broader Python ecosystem; participants familiar with ISLA in particular perceived a lower barrier for defining semantic constraints. These lessons shaped the released tool: ready-to-run examples, a simple CLI, and a Python API support quick iteration with little setup overhead.

IV. Related Work

Automatically obtaining inputs that cover the behavior of a given program is a long-standing challenge in computer science. We survey the main approaches below.

1) Random inputs and evolution: Miller's fuzzing work [7] demonstrated that random byte sequences suffice to expose severe bugs in input parsers, and simple random fuzzing remains a standard robustness test. Search-based testing [8] extends this by mutating a population of *seed inputs* and retaining those that improve coverage. AFL [9] and LIBFUZZER [10] implement this strategy with high efficiency and have uncovered thousands of bugs in open-source software. Effectiveness nonetheless hinges on *seed diversity*: features absent from the seeds are unlikely to emerge through mutation.

2) Grammars and language learners: For complex input languages, a *language specification* is indispensable. Context-free grammars for test generation have proven highly effective [11]: LANGFUZZ [12] uncovered over 2,600 bugs in JavaScript interpreters; NAUTILUS [13] pairs grammar-based generation with coverage feedback; and AFLSMART [14] combines the structural awareness of PEACH [15] with the mutation engine of AFL. Because writing grammars can be nontrivial, *language learners* such as ARVADA [16] and MIMID [17] infer them from example inputs or program behavior, while Bettscheider et al. [18], [19] leverage symbolic analysis without samples; ISLEARN [5] additionally learns semantic constraints from valid input sets.

3) Constraint solving: Symbolic test generators collect the *path constraints* that must hold to reach a given program location, then use *model checkers* and *constraint solvers* to find satisfying inputs, as in KLEE [20] and SAGE [21]. Scalability is a persistent obstacle, however: even a simple arithmetic expression parser can induce intractable constraints [22]. Practical tools therefore pair constraint solving with seed inputs [23], evolutionary search [24], or grammars [25], or substitute lightweight alternatives such as gradient descent [26].

4) Domain-specific generators and constraints: Context-free grammars cannot enforce semantic properties such as identifier scoping or checksum consistency. Domain-specific generators like CSMITH [27], TLS-Attacker [28], and MORPHEUS [29] handle such requirements, but represent substantial engineering investments. ISLA [5] addresses expressiveness by augmenting grammars with SMT-LIB [30] constraints solved via Z3 [31]. In a recent fuzzer comparison [32], ISLA achieved 100% input validity across all subjects, yet its SMT-based approach also yielded the lowest throughput.

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

V. Limitations and Future Work

In the future, we will tackle the following limitations:

a) Execution feedback: Since generation is purely specification-driven, FANDANGO does not observe the program under test, and therefore cannot steer evolution towards unexplored behavior. Incorporating execution feedback (e.g., execution features [33]) as an additional fitness term is a natural next step that would complement the existing constraint-based fitness without requiring changes to the specification language.

b) State awareness: FANDANGO generates individual inputs in isolation, without awareness of stateful interactions or protocol sequences. Extending the grammar formalism to model state machines and generating sequences of inputs that satisfy inter-input constraints would enable FANDANGO to target a broader class of applications, such as network services or interactive programs. An extension that is already publicly available allows FANDANGO to encode protocol state machines [34], enabling the generation of valid sequences of messages for protocol fuzzing.

c) Specification burden: Effective use of FANDANGO requires a grammar that accurately models the target input format, which can be a substantial effort for complex or binary formats. Grammar inference from existing samples and LLM-assisted grammar synthesis are promising directions we intend to explore to lower this barrier, similar to AUTOSPEC [6].

VI. Conclusion

We presented FANDANGO, a language-based test generator that replaces symbolic constraint solving with heuristic optimization techniques. By evolving a population of syntactically valid derivation trees against Python-expressed constraints, FANDANGO achieves one to three orders of magnitude higher throughput than ISLA without sacrificing precision, while giving testers the full expressive power of the Python ecosystem. The tool is open-source, installable with a single

[2]https://cms.cispa.saarland/fuzzing25/

[3]https://rio2025.dc.exa.unrc.edu.ar/

`pip` command, and comes with a command-line interface, a programmatic API, and ready-to-run examples. We believe FANDANGO is a step further to lowering the barrier to language-based testing and invites the community to extend it towards their specific needs.

FANDANGO is available as an open-source project under the European Union Public License (EUPL) v1.2. The source code, installation instructions, and example datasets can be found on our GitHub repository:

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

REFERENCES

[1] H. Al Salem, "A review on grammar-based fuzzing techniques," 2021.

[2] D. Steinhöfel and A. Zeller, "Language-based software testing," *Communications of the ACM*, vol. 67, no. 4, p. 80–84, March 2024. [Online]. Available: https://doi.org/10.1145/3631520

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

[4] T. Bartz-Beielstein, J. Branke, J. Mehnen, and O. Mersmann, "Evolutionary algorithms," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 4, no. 3, pp. 178–195, 2014.

[5] 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, p. 583–594. [Online]. Available: https://doi.org/10.1145/3540250.3549139

[6] K. Liu, D. Chakraborty, A. Liggesmeyer, and A. Zeller, "Synthesizing precise protocol specs from natural language for effective test generation," 2025. [Online]. Available: https://arxiv.org/abs/2511.17977

[7] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, December 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[8] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 153–163.

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

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

[11] W. H. Burkhardt, "Generating test programs from syntax," *Computing*, vol. 2, no. 1, pp. 53–73, Mar 1967. [Online]. Available: https://doi.org/10.1007/BF02235512

[12] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *USENIX Security Symposium*. Bellevue, WA: USENIX Association, 2012, pp. 38–38. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[13] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," 01 2019. [Online]. Available: https://doi.org/10.14722/ndss.2019.23412

[14] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 9, pp. 1980–1997, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2019.2941681

[15] "Peach fuzzer, community edition," https://gitlab.com/peachtech/peach-fuzzer-community, retrieved 2022-02-12.

[16] N. Kulkarni, C. Lemieux, and K. Sen, "Learning highly recursive input grammars," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, pp. 456–467. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678879

[17] 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

[18] 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*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 522–526. [Online]. Available: https://doi.org/10.1145/3663529.3663790

[19] ——, "Inferring input grammars from code with symbolic parsing," *ACM Trans. Softw. Eng. Methodol.*, November 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3776743

[20] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: https://dl.acm.org/doi/10.5555/1855741.1855756

[21] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, January 2012. [Online]. Available: http://doi.acm.org/10.1145/2090147.2094081

[22] N. Bjørner, "Z3 fails to terminate despite set timeout," https://github.com/Z3Prover/z3/issues/5891#issuecomment-1063110825, March 2022.

[23] K. Sen, D. Marinov, and G. Agha, "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)*. ACM, 2005, pp. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[24] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Network and Distributed System Security Symposium (NDSS)*, vol. 16, 2016, pp. 1–16. [Online]. Available: https://sites.cs.ucsb.edu/~vigna/publications/2016_NDSS_Driller.pdf

[25] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2008, pp. 206–215. [Online]. Available: https://doi.org/10.1145/1375581.1375607

[26] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018, pp. 711–725. [Online]. Available: https://doi.org/10.1109/SP.2018.00046

[27] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ser. PLDI '11. ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

[28] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1492–1504, code available at https://github.com/tls-attacker/TLS-Attacker. [Online]. Available: https://doi.org/10.1145/2976749.2978411

[29] M. Yahyazadeh, S. Y. Chau, L. Li, M. H. Hue, J. Debnath, S. C. Ip, C. N. Li, E. Hoque, and O. Chowdhury, "Morpheus: Bringing the (PKCS) one to meet the oracle," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2474–2496.

[30] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," 2016. [Online]. Available: https://smt-lib.org

[31] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[32] S. M. Hasan, P. Kozyreva, and E. Hoque, "FuzzEval: Assessing fuzzers on generating context-sensitive inputs," 2024. [Online]. Available: https://arxiv.org/abs/2409.12331

[33] M. Smytzek, M. Eberlein, L. Grunske, and A. Zeller, "How execution features relate to failures: An empirical study and diagnosis approach," *ACM Trans. Softw. Eng. Methodol.*, December 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3783989

[34] A. Liggesmeyer, J. A. Z. Amaya, and A. Zeller, "Language-based protocol testing," 2026. [Online]. Available: https://arxiv.org/abs/2509.20308