Closing Semantic Gaps in ESBMC-Python: Semantic Extensions for Bounded Model Checking

Sergillam B. Oliveira[†‡], Eddie de Lima Filho[†‡], and Lucas C. Cordeiro[‡§*]
† *TPV Technology, Manaus, Brazil*
‡ *Federal University of Amazonas, Manaus, Brazil*
§ *The University of Manchester, Manchester, UK*
{sergillam.oliveira,eddie.filho}@tpv-tech.com, lucas.cordeiro@manchester.ac.uk

***Abstract*—Python is widely used in scientific computing, machine learning, and production software, increasing the need for reliable verification of its programs. Formal verification is challenging because Python constructs must be translated into a verification-oriented intermediate representation without changing their observable behavior. Unlike contract-based or dynamic-symbolic tools such as Nagini, CrossHair, and PyExZ3, ESBMC-Python lowers Python programs into ESBMC's GOTO-based bounded model checking pipeline, where semantic mismatches in arithmetic, strings, complex numbers, and exception handling can produce spurious behaviors or suppress behaviors that exist in the original Python program. This paper extends the previous ESBMC-Python front-end with semantic translation strategies for strings, mathematical operations, and complex numbers. The extended front-end processes 43 of 164 HumanEval benchmarks previously blocked by unsupported string operations and increases the studied standard-library surface to 42/47 str methods, 56/61 math functions, and 23/26 cmath functions. Regression tests and representative CPython divergence cases provide evidence that the proposed translations preserve the intended Python behavior for the evaluated semantic subset.**

***Index Terms*—Formal Verification, Bounded Model Checking, ESBMC, Python, SMT, Semantic Fidelity.**

I. Introduction

Python is widely used in critical domains such as scientific computing and machine learning, making its formal verification increasingly relevant. While testing techniques such as fuzzing exercise selected execution paths [1], [2], Bounded Model Checking (BMC) exhaustively explores program behaviors within explicit bounds [3]. Verifying dynamic languages, however, introduces additional architectural challenges [4]. Since SMT solvers operate on an intermediate representation (IR) rather than the original source program, semantic mismatches introduced during front-end translation may cause the verifier to analyze behaviors that differ from Python execution.

Several tools have advanced Python verification, though they operate on different paradigms and abstract semantic details differently. For instance, Nagini [5] provides robust static verification but relies on user-provided contracts mapped to the Viper infrastructure. Conversely, dynamic symbolic execution and analysis tools such as PyExZ3 [6], CrossHair [7], and pySim [8] explore program paths dynamically. Because these approaches use different verification abstractions, they do not face the same AST-to-GOTO translation boundary considered in this work, where Python-specific arithmetic and exception semantics must be explicitly preserved.

Prior work introduced native AST-to-IR BMC for Python in ESBMC-Python [4] and explored LLM-assisted translation for bounded verification [9], while recent ESBMC developments improved the underlying verification infrastructure [10]. However, these approaches do not specifically address semantic divergences introduced when Python constructs are lowered to a lower-level GOTO representation, where arithmetic, exceptions, strings, and complex values may exhibit different behavior.

Therefore, the central problem addressed in this work is not merely unsupported Python functionality, but semantic fidelity during translation: the generated IR must represent the behavior of the original Python program rather than that of an approximate lower-level counterpart. In this work, semantic fidelity means preserving the observable result or corresponding Python exception for the evaluated operations under the supported bounded representation.

To address these limitations, this work extends the native ESBMC Python front-end with explicit translation strategies that preserve Python-specific behavior while reusing the existing symbolic-execution and SMT infrastructure. The contributions are:

- A bounded string translation strategy supporting Python-specific operations, exception paths, and symbolic strings.
- Python-aware arithmetic translation rules preserving selected modulo, division, domain-error, and type-sensitive exception behavior.
- An IR encoding for Python complex numbers through real-imaginary floating-point components and selected cmath operations.

As an implementation-level optimization, the front-end also employs a structural conversion cache that reuses conversion decisions without storing symbolic-execution state or solver-dependent results.

The remainder of this paper is organized as follows. Section II introduces the theoretical background. Section III presents the proposed front-end architecture and implementation details. Section IV reports the experimental evaluation. Finally, Section V concludes the paper.

II. Background

This section briefly introduces Bounded Model Checking (BMC), ESBMC, and the semantic role of its Python front-end.

A. Bounded Model Checking and ESBMC

Bounded Model Checking (BMC) searches for property violations within bounded executions by encoding verification conditions as satisfiability problems [3]. ESBMC translates programs into a GOTO representation, applies symbolic execution, and encodes the resulting conditions for SMT solving satisfiable violations yield counterexamples [3], [11].

B. ESBMC Python Front-end and Semantic Challenges

The ESBMC Python front-end derives type information from the Python AST and translates expressions and statements into the GOTO representation consumed by the existing symbolic-execution and SMT pipeline [4]. Because subsequent verification operates on this representation rather than the original source, semantic translation errors may cause the analyzed behavior to differ from Python execution.

III. Implementation Details

The proposed semantic extensions are implemented in the C++ ESBMC Python front-end [4][1]. Fig. 1 situates the proposed front-end extensions within the broader Python processing flow, from source-code preparation to the generation of ESBMC's GOTO representation. The contributions of this work are confined to the front-end conversion stage, the subsequent symbolic-execution and SMT-solving stages remain unchanged.

A. Architecture and Conversion Flow

Fig. 1 shows the broader processing path from *Python Source Code* through the standard Python compilation context, in which bytecode may be materialized as a `.pyc` file or maintained in memory, before reaching *Front-end Core Processing*. The semantic extensions proposed in this work operate inside this front-end core. From the *Entry Point*, the *Conversion Engine* traverses the Python program representation through the *AST Iteration Loop*, while *Instruction Dispatch* selects the translation procedure for each supported construct. Domain-specific operations are routed to *Extend Module Support*, which contains the *Math*, *String*, and *Complex Numbers* extensions. The *Convert instruction* stage then produces ESBMC-compatible intermediate expressions, either directly or through C or Python operational models. Finally, *Create Goto Functions* incorporates the translated instructions into the *GOTO Program Representation* consumed by the existing verification pipeline.

To illustrate the front-end path highlighted in Fig. 1, consider `assert -3 % 2 == 1`. Once the program reaches the front-end conversion stage, its AST representation contains an `Assert` node whose predicate is a modulo expression. During AST traversal, the front-end dispatches this expression to the Math extension, because Python modulo semantics differ from C remainder semantics for operands with different signs. The extension lowers the operation using the Python-compatible formulation $x - \lfloor x/y \rfloor y$, which makes the remainder follow the sign of the divisor. In the generated

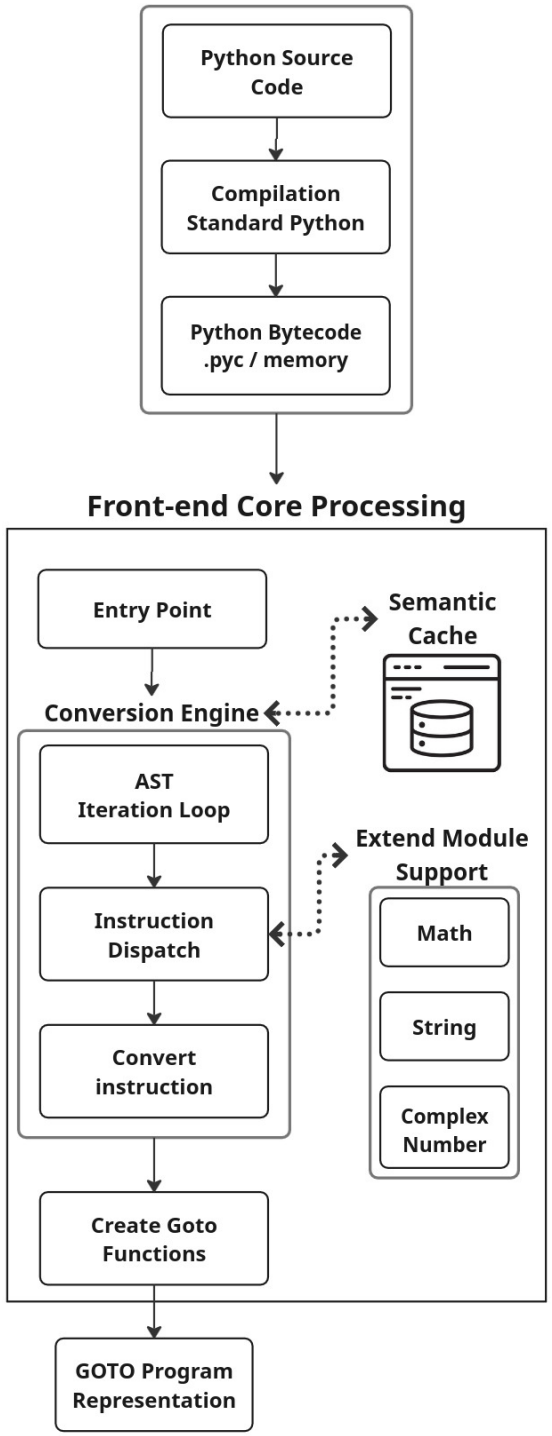

Fig. 1. Python processing flow and ESBMC front-end components used by the proposed semantic extensions.

[1]https://github.com/esbmc/esbmc/

GOTO program, this expression appears as the assertion condition, together with the explicit `ZeroDivisionError` path for modulo by zero. Symbolic execution then generates the corresponding verification conditions, ESBMC proves the generated verification conditions for this input, yielding `VERIFICATION SUCCESSFUL`. Thus, although the surrounding Python preparation flow remains unchanged, the front-end conversion ensures that the generated GOTO property reflects Python modulo semantics rather than the remainder semantics of a lower-level approximation.

The *Semantic Cache* interacts with the *Conversion Engine* during front-end traversal to reuse structural conversion decisions. As indicated by its lateral connection in Fig. 1, the cache is a cross-cutting front-end mechanism rather than a separate stage in the verification pipeline. The following subsections describe the domain-specific translation strategies and the cache's contents, lifetime, and reuse policy.

B. String Operations

The String extension uses bounded null-terminated character arrays and targets ASCII-oriented behavior within ESBMC-Python's string model[2]. The evaluated model covers bounded ASCII-oriented strings unrestricted Unicode transformations, normalization, embedded NUL handling, and arbitrary string lengths remain outside its scope.

To preserve the relevant Python semantics for the supported string operations, the module adopts the following translation strategies:

- **Immutability Preservation:** In Python, strings are immutable. Consequently, operations that transform strings (e.g., slicing, case conversion) never modify the original array. Instead, the module constructs a new array in the IR, preserving the symbolic representation of the original string.
- **Operational C Models:** Complex string methods, such as `upper()`, `replace()`, and `strip()`, are translated into executable C models. These models are subsequently processed by the existing symbolic execution engine, allowing their behavior to participate in the generated verification conditions.
- **Explicit Exception Modeling:** Runtime errors must also remain explicit in the translated program to avoid silently accepting executions that would raise exceptions in Python. The front-end introduces strict bounds checks for string indexing. Out-of-bounds accesses are deliberately represented as `IndexError` conditions in the IR rather than being silently ignored. Similarly, `ValueError` and `TypeError` are represented for supported operations whose arguments or values violate the corresponding Python semantics.

The module also supports symbolic strings whose concrete values are not fixed before verification. The front-end uses a bounded string model: the shared runtime bound is 256 bytes for the main string-length helpers, although individual operational models may impose smaller method-specific bounds, as defined in their corresponding models. Therefore, the evaluation does not assume that every string method has exactly the same operational limit. Inputs outside the supported bounds are not treated as evidence of complete CPython string semantics and must not be silently generalized from the evaluated cases.

C. Mathematical Operations

The verification of numerical Python programs requires the front-end representation to preserve Python arithmetic semantics[3], which differ from C semantics for several operators used by the existing verification infrastructure. The Math extension models Python-specific rules directly in the front-end, including modulo sign behavior, floor-division rounding, domain errors, and exception paths for unsupported argument types.

The module enforces precision in unary operations. For primitive numeric types, the front end preserves the operand type when constructing unary expressions, such as negations, to avoid unintended type promotions.

The module also resolves semantic divergences in binary arithmetic. The Python modulo operator (`%`) produces a remainder whose sign follows the divisor, whereas the corresponding C remainder follows the dividend. The front end models this algebraically as $x - \lfloor x/y \rfloor \cdot y$. Similarly, Python floor division (`//`) rounds towards negative infinity. The module applies a sign correction in the IR when operands have opposite signs and a nonzero remainder. For true division (`/`), floor division (`//`), and modulo (`%`), a zero divisor is translated into the corresponding `ZeroDivisionError` path rather than being left to lower-level arithmetic or solver-specific behavior.

Furthermore, exponentiation involving negative or floating point exponents is routed to the C `pow` function with operands promoted to `double`. This avoids erroneous integer truncation to zero for cases such as `2 ** -1`. When integer operands are promoted to `double`, exact preservation is limited to magnitudes no greater than 2^{53}, the range in which IEEE 754 binary64 represents every integer exactly. Values outside this range may lose integer precision and are treated as a bounded-model limitation rather than as full CPython arbitrary-precision integer semantics. Finally, the translation of `round()` accounts for Python's round half to even behavior rather than directly adopting the corresponding C rounding semantics.

The implementation introduces explicit domain and type checking to prevent silent failures. Directly mapping Python mathematical functions to their C counterparts can change error behavior for example, an invalid Python domain operation may be represented as a floating point special value instead of raising the corresponding Python exception. The front end injects explicit domain checks, so that operations such as `math.sqrt(-1.0)` are represented with the corresponding `ValueError` behavior. These checks also apply to symbolic operands: for `math.sqrt(x)`, the generated IR preserves

[2]https://docs.python.org/3/library/stdtypes.html

[3]https://docs.python.org/3/library/math.html

both the error path in which `x < 0` and the normal path in which `x >= 0`, rather than relying only on constant folding. In addition, the front-end detects complex arguments supplied to real-valued math functions and represents the corresponding `TypeError` before generating the normal translation path. This interaction with the complex number support is discussed further in the following subsection.

To support the math module, the architecture implements a three-tiered resolution strategy. Constant expressions are evaluated during front-end conversion when their operands are known. Symbolic operands with direct equivalents are dispatched to the standard C mathematical library. For specialized functions lacking a direct C equivalent, such as `math.gcd` and `math.factorial`, the front end routes execution to Python operational models that are translated by the existing ESBMC-Python front-end. Unlike CPython integers, which are arbitrary precision, ESBMC-Python represents primitive integers using the configured target integer widths. Arithmetic exceeding those widths follows the verifier's machine-integer encoding for the selected target consequently, such results are not generalized to CPython's unbounded integer semantics.

D. Complex Numbers

Complex values are represented in the IR as pairs of `double`-precision floating-point components corresponding to their real and imaginary parts[4]. This representation maps each Python complex value to two `double`-precision components, enabling supported complex operations to be reduced to existing floating-point expressions without introducing a solver-level complex-number theory.

Arithmetic operations are expanded into expressions over the real and imaginary components. For example, multiplication of $(a+bi)$ and $(c+di)$ is encoded as $(ac-bd)+(ad+bc)i$. Exception propagation is handled before values are wrapped as complex objects, preventing error states from being treated as ordinary complex values.

Unary negation also requires component-wise treatment. To preserve the distinction between $+0.0$ and -0.0 under IEEE 754 semantics, the real and imaginary components are negated using explicit floating point subtraction from zero rather than relying on a generic unary transformation.

The module also models the supported forms of the Python `complex()` constructor. The front-end handles zero-argument construction, promotion from real values, supported string representations, and the two-argument case, including cases in which one or both arguments are themselves complex values. Furthermore, during type promotion and arithmetic operations, the implementation checks whether an operand already represents an exception state, such as `TypeError` or `ValueError`. Such exceptions are propagated rather than being encapsulated as ordinary complex values, preserving the corresponding error path in the translated program.

The implementation provides Python operational models for the cmath module covering 23 functions. For selected operations and argument patterns, the front-end provides direct IR translation paths, whereas the remaining supported cases are delegated to those Python models.

The complex-number support also interacts with the math module by detecting complex arguments supplied to functions defined over real values. When such an argument is identified, the front-end represents the corresponding `TypeError` before generating the normal translation path for the mathematical operation.

E. Semantic Cache

The translation of a Python program into the GOTO IR is an iterative process that repeatedly resolves identifiers, infers types, and navigates inheritance hierarchies. Without a reuse mechanism, the front-end may repeatedly evaluate equivalent structural queries during a single conversion. The Semantic Cache is therefore used as an implementation-level optimization and is orthogonal to the semantic extensions evaluated in this paper.

The Semantic Cache stores only conversion-time structural facts, such as method-existence queries and class-inference results. Cached entries depend only on structural conversion information and never on symbolic values, path conditions, or solver results. The cache is cleared at the beginning of each front-end conversion and is not shared across independent verification runs.

Two additional design decisions govern cache reuse and termination:

- **Negative Caching:** The cache explicitly stores negative results, such as the absence of a method within a class hierarchy. Without negative caching, repeated queries that produce no match would require the same structural traversal to be performed again.
- **Recursion Prevention:** To guarantee termination in the presence of cyclic structural dependencies, the cache employs a provisional negative-marking strategy. Provisional negative entries are used exclusively as recursion guards while a query is active. Such an entry may terminate a recursive lookup for the same key, but it is not propagated as a final result to unrelated upstream queries. When the original lookup completes, the provisional value is replaced by the computed result, if conversion terminates abnormally, the entry is discarded with the conversion context. This invariant prevents temporary recursion-breaking values from becoming definitive structural facts.

By reusing conversion-time structural decisions, the cache reduces repeated front-end work without introducing dependencies on symbolic verification state. Its performance characteristics are outside the experimental scope of this paper, which focuses on the semantic extensions.

IV. Experiments

This section evaluates the proposed Python front-end extensions with respect to semantic coverage and semantic fidelity for representative Python cases. The evaluation focuses on the String, Math, and Complex Numbers extensions and does not claim complete preservation of CPython semantics.

[4]https://docs.python.org/3/c-api/complex.html

A. Research Questions

The experimental evaluation is organized around two research questions that assess the proposed extensions from complementary perspectives: semantic coverage and semantic fidelity.

RQ1: Semantic Coverage. *To what extent do the proposed extensions increase the set of Python constructs and programs processable by ESBMC compared with the previous front-end?* This question evaluates the practical increase in front-end coverage provided by the String, Math, and Complex Numbers modules. We consider the regression suites associated with the implemented features and external Python benchmarks to determine which programs can be successfully translated and processed by the extended front-end.

RQ2: Semantic Fidelity. *Do the proposed translations reproduce CPython behavior for representative semantic divergence cases exercised by the regression suite?* This question evaluates whether the implemented translation rules reproduce CPython behavior in cases susceptible to semantic divergence under lower-level GOTO translation. We consider arithmetic rules, domain errors, exception behavior, string operations, and complex-number arithmetic, using CPython as the behavioral reference.

B. Setup

We conducted all experiments on a Dell Inspiron 13 equipped with an Intel Core i7-1360P processor, 16 GB of RAM, and Ubuntu 24.04. The reference implementation of the extended Python front-end is integrated into the open-source ESBMC repository[5].

We ran ESBMC 8.5.0 at commit `7152d23b86e3d06a9dd303e4c3cce52d6418a8ad`, CPython 3.12.3, and Z3 4.13.3 [12].

At the evaluated commit, the Python regression suite contains 5,169 tests. The extension-specific recut used in this paper identifies 556 string-related, 226 math-related, and 140 complex/cmath regression tests. Tests are assigned to extension-specific groups according to their regression-suite location and primary Python feature exercised.

These counts describe the regression-suite coverage available at the evaluated commit. HumanEval contributes a separate public set of 164 tasks. A HumanEval task is classified as *processable* when ESBMC-Python parses it, translates it into a GOTO program, and reaches the verification pipeline under the reported configuration.

To evaluate the impact of the proposed extensions, we use the prior ESBMC-Python implementation [4] as the implementation baseline. CrossHair and Nagini have not been executed on the same subset, so the paper does not claim a direct comparative result against those tools. Such a comparison requires an additional experiment with aligned specifications and supported inputs.

[5]https://github.com/esbmc/esbmc/

C. Results

The results are organized according to the two research questions defined in Section IV, covering semantic coverage and semantic fidelity with respect to the evaluated Python subset.

RQ1: Semantic Coverage. Table I reports the supported API surface and the regression-test effort for the three semantic extension modules as separate metrics. The API denominators refer to the functions considered in this study, whereas the test counts indicate the number of regression cases used to validate the implementations.

TABLE I
SUPPORTED API SURFACE AND REGRESSION COUNTS AT THE EVALUATED COMMIT

Extension	Regression Tests	Supported API Surface
String	556	42/47 methods 8 predicates 11 operators
Math	226	Python arithmetic 56/61 math functions
Complex	140	Complex arithmetic 23/26 cmath functions

Table I separates implemented API surface from regression-test effort. The former measures supported standard-library surface, while the latter measures validation coverage at the evaluated commit.

Compared with the previous ESBMC-Python baseline, the extended front-end enabled 43 of 164 HumanEval benchmarks that were previously blocked by unsupported string operations. The available evidence does not provide the total baseline and extended processability counts or a complete classification of the remaining blocked tasks, so those quantities are not reported.

RQ2: Semantic Fidelity. Semantic fidelity is assessed through representative regression cases that exercise known divergences between Python semantics and direct lower-level mappings, as summarized in Table II. The repository additionally contains a differential Python-model test infrastructure based on `pytest` and Hypothesis.

Table II summarizes representative semantic divergences addressed by the proposed extensions, including mixed-sign modulo and floor division, explicit `ValueError` and `IndexError` paths, and complex multiplication. These cases illustrate how Python-aware translation prevents lower-level semantics from changing the behavior represented in the generated GOTO program.

D. Discussion

The results provide evidence for the evaluated subset of Python semantics rather than complete CPython equivalence.

TABLE II
REPRESENTATIVE SEMANTIC PROPERTIES

Case	CPython	Model	Semantic divergence
`-3 % 2`	`1`	`1`	remainder sign
`-3 // 2`	`-2`	`-2`	floor vs. truncation
`sqrt(-1)`	`Value Error`	error path	exception vs. special value
`"abc"[5]`	`Index Error`	error path	bounds exception
$(1+2i)(3+4i)$	$-5+10i$	$-5+10i$	algebraic complex expansion

The String, Math, and Complex Numbers extensions address complementary classes of translation gaps: string behavior is encoded through bounded operational models, arithmetic divergences are corrected during IR construction, and complex values are reduced to supported real-imaginary floating-point components. Across these mechanisms, Python-specific exception behavior remains explicit in the generated program, preventing lower-level representations from silently changing the reachable behavior analyzed by ESBMC.

The Semantic Cache remains orthogonal to these semantic mechanisms: it reuses structural front-end decisions but does not alter symbolic values, path conditions, or solver reasoning. It is therefore treated as an implementation optimization rather than as an independent semantic contribution.

E. Threats to Validity

The benchmark set consists of ESBMC regression tests and programs from the 164 task HumanEval benchmark. These workloads exercise relevant Python front-end behavior, but they do not represent the full Python ecosystem, third-party libraries, unrestricted Unicode text processing, or arbitrary scientific-computing programs.

API coverage counts measure implemented surface rather than complete semantic equivalence. Similarly, processability indicates whether a benchmark reaches the verification pipeline but does not by itself establish semantic equivalence with CPython. The representative cases used for RQ2 cover selected divergence patterns rather than the complete Python language. In addition, the evaluation does not measure solver scalability as a function of symbolic string length broader Unicode semantics and unrestricted Python integer behavior also remain outside the evaluated scope.

Because part of the regression suite was developed alongside the implementation, test-design bias cannot be excluded HumanEval provides an external benchmark that partially mitigates this threat.

V. CONCLUSION

This work extends ESBMC-Python with explicit semantic modeling for strings, arithmetic, and complex numbers. The proposed extensions increase the supported Python standard-library surface and enable 43 HumanEval tasks previously blocked by unsupported string operations. Representative semantic divergence cases show that Python-aware translation is necessary to preserve arithmetic behavior, exception paths, string semantics, and complex-number operations when lowering programs into GOTO-based verification infrastructure. These results support treating source-language semantics as a front-end responsibility in bounded model checking while reusing the existing symbolic-execution and SMT backend.

ACKNOWLEDGEMENT

The results presented in this paper were sponsored by ENVISION Indústria de Produtos Eletrônicos LTDA, under the terms of Brazilian federal law No. 8.387/91 (SUFRAMA). This research was conducted by Envision (TPV Group).

REFERENCES

[1] F. Izumi, E. B. de Lima Filho, L. C. Cordeiro, O. B. Maia, R. Fabrício, B. Farias, and A. Silva, "A fuzzing-based test-creation approach for evaluating digital tv receivers via transport streams," *Softw. Test. Verification Reliab.*, vol. 33, no. 1, 2023.

[2] S. B. Oliveira, E. B. de Lima Filho, O. B. Maia, and L. C. Cordeiro, "Automated evaluation of digital tv middleware: Using fuzzing and pairwise for test generation," in *IEEE International Conference on Consumer Electronics, ICCE 2025, Las Vegas, NV, USA, January 11-14, 2025*, pp. 1–5, IEEE, 2025.

[3] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2012.

[4] B. Farias, R. Menezes, E. B. de Lima Filho, Y. Sun, and L. C. Cordeiro, "Esbmc-python: A bounded model checker for python programs," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1836–1840, 2024.

[5] M. Eilers and P. Müller, "Nagini: a static verifier for python," in *International Conference on Computer Aided Verification*, pp. 596–603, Springer, 2018.

[6] T. Ball *et al.*, "Pyexz3: An exploration of dynamic symbolic execution for python." https://github.com/thomasjball/PyExZ3, 2015.

[7] P. Schanely and contributors, "Crosshair: An analysis tool for python." https://github.com/pschanely/CrossHair, 2020.

[8] pySym Contributors, "pysym: Python symbolic execution." https://github.com/bannsec/pySym, 2017.

[9] S. Shivaji, K. Havelund, N. Lobakhina, L. C. Cordeiro, and A. Pinto, "Llm-assisted translation and bounded model checking of python code," in *International Symposium on AI Verification (AISoLA)*, 2026.

[10] T. Wu, X. Li, E. Manino, R. S. Menezes, M. R. Gadelha, S. Xiong, N. Tihanyi, P. Petoumenos, and L. C. Cordeiro, "Esbmc v7.7: Efficient concurrent software verification with scheduling, incremental smt and partial order reduction," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pp. 223–228, Springer, 2025.

[11] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Esbmc 5.0: an industrial-strength c model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 888–891, 2018.

[12] 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*, pp. 337–340, Springer, 2008.