Improving Automated Program Verification for Java Programs with Fuzzing

Soha Hussein
Department of Information Systems
Ain Shams University
Cairo, Egypt
soha.hussein@cis.asu.edu.eg

Stephen McCamant
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, MN, USA
mccamant@cs.umn.edu

***Abstract*—Formal verification of programs has long been used to provide rigorous correctness guarantees on program behavior. However, the scalability of these techniques is limited, as they require appropriate encoding for various program constructs—encodings that are often unavailable in practice, or may grow in complexity, making them difficult to solve using these techniques.**

On the other hand, coverage-guided fuzzing (CGF) is a random testing technique that uses program coverage to select and prioritize inputs. While it can scale to more complex programs—as it does not require any state encoding—this very characteristic makes it fall short in providing correctness guarantees, as it is, by definition, an incomplete approach.

In this paper, we propose improving automated program verification for Java programs using program fuzzing to find counterexamples. More precisely, we adapt a coverage-guided fuzzing tool to be used on verification tasks from the top software verification competition (SV-COMP). Our results show the advantages of using fuzzing in verification, either due to the difficulty or absence of appropriate verification summarization for certain program functions, or when path/state explosion becomes a bottleneck.

***Index Terms*—program verification, fuzzing, symbolic execution, testing, program analysis**

I. Introduction

Formal verification of programs uses formal methods to prove or disprove the correctness of a property in a program under test. Such verification techniques are widely used in safety-critical software systems and, in recent years, have been adopted in larger-scale industrial software. For example, Amazon has employed pioneering software verification researchers in its AWS Automated Reasoning Group to provide high-assurance security at scale for its products [1], [2]. Similarly, formal methods have been used to verify Kubernetes controllers in Google clusters [3]. With the current wave of AI and the growing need to validate and verify its outputs, new research has emerged focusing on developing methods and tools for verifying AI components, such as neural network classifiers and reinforcement learning policies [4].

In formal verification, programs are usually modeled as a finite-state machine, where the nodes represent the states of a system and the edges represent possible transitions between states. In Bounded Model Checking (BMC), the program model is constructed by unrolling the program into a fixed number of iterations (i.e., states). The property to be checked is then conjoined with the bounded model of the program, and the result is checked for satisfiability using Satisfiability Modulo Theories (SMT) or SAT solvers. Symbolic execution (SE) can be considered as a bounded model checking technique that is based on program analysis, where the programs are analyzed by executing them dynamically while using symbols for program inputs instead of concrete values.

These techniques, while they provide assurance over the correctness of programs, require heavy machinery for translating the program into an appropriate logical formula, in addition to the computational expense of checking the satisfiability property—often an expensive task. These factors limit the scalability of such techniques to large and complex programs.

On the other hand, Coverage-Guided Fuzzing (CGF) is an incomplete random testing approach that can be easily scaled, as it does not require the heavy machinery needed in formal verification. In CGF, the program under test is executed with random inputs that are prioritized based on whether they achieve new program coverage. That is, when a particular input leads to new coverage, it is identified as interesting and used as the basis for generating subsequent inputs. These new inputs are produced by mutating the bits of the interesting input.

One important feature that verification techniques provide is that when they fail to prove a property, they typically generate a counterexample. A counterexample illustrates a concrete execution path or input that violates the property under consideration, demonstrating precisely why the property does not hold. This is crucial for the verification process, since counterexamples guide the next steps: either refining or correcting the property, or modifying the program itself. In this way, counterexamples enable an iterative cycle of improvement, allowing developers and researchers to make tangible progress. By contrast, if verification terminates with a timeout or inconclusive result without producing a counterexample, it becomes far less actionable—leaving no clear direction for what to fix or how to proceed.

In this paper, we argue that coverage-guided fuzzing can complement verification techniques by improving the reachability of property falsification in Java programs. By producing concrete failing inputs that act as counterexamples, fuzzing makes the verification feedback cycle more efficient and actionable. More precisely, we show how we modified the usage

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00037

```
1.void checkPattern(String input) {
2.  if (input.length() < 7)
3.    return;
4.  if (input.charAt(0) == 'f')
5.    if (input.charAt(2) == 'a')
6.      if (input.charAt(4) == 'i')
7.        if (input.charAt(6) == 'l')
8.          assert false;
9.  System.out.println("Pattern NOT found");}
```

Fig. 1: Example of a program checking whether the input contains the word `fail` in every other character

of the popular coverage-guided fuzzer AFL [5], to be applied in the context of verification tasks. Since AFL is traditionally designed to fuzz C programs, we built on an extension called JQF-AFL [6], which is an existing extension that enables AFL to fuzz Java programs. This is achieved by spawning a proxy between the Java program and AFL, allowing the exchange of coverage information and new inputs between AFL and the Java program.

To evaluate the modified JQF-AFL extension, we tested it on 420 unsafe verification tasks from SV-COMP 2025. A verification task consists of a program and a property, and we call it unsafe if the program fails to satisfy the property. Our results show that the fuzzer was able to reach the unsafe statement in 281 of these tasks, including 3 that were not found by any of the verification tools. Our results highlights the advantage of using a fuzzer when the verification techniques face limitation from having to explore a large state space, i.e. path/space explosion.

II. Example of Verification and Fuzzing

To illustrate the differences between symbolic execution and fuzzing, we use a simple program that checks whether its input matches a specific pattern. Symbolic execution systematically reasons about program paths using logical constraints, while fuzzing explores them through randomized inputs and coverage feedback.

Figure 1 shows a program that attempts to check whether the input contains the word `fail` at every other character. To verify this program using symbolic execution, the verification tool systematically explores all possible execution paths while constructing logical constraints that capture the conditions on input variables and checking their satisfiability.

For example, in Figure 1, there are six execution paths – one that reaches the `assert` statement (**line 8**) and others that represent incomplete patterns. Listing 1 shows the constraint generated to check the path that reaches the `assert` statement. This logical constraint is then passed to the solver to check whether there exists a satisfiable assignment. The constraint begins by specifying the string logic to be used by the solver (**line 1**). Then, the input variable is declared (**line 2**), and constraints are placed on its pattern (**lines 3–7**). Finally, (**lines 8–9**) check whether the constraint is satisfiable and, if so, instruct the solver to print the model – a satisfying assignment to the characters in the `input` variable.

```
1.(set-logic QF_S)
2.(declare-fun input () String)
3.(assert (>= (str.len input) 7))
4.(assert (= (str.at input 0) "f"))
5.(assert (= (str.at input 2) "a"))
6.(assert (= (str.at input 4) "i"))
7.(assert (= (str.at input 6) "l"))
8.(check-sat)
9.(get-model)
```

Listing 1: SMT-LIB encoding of the path to `assert false`

On the other hand, when testing this program using coverage-guided fuzzing, the fuzzer does not create any logical constraints. Instead, the fuzzer starts with random values for the `input` variable. If new coverage is achieved, the random input used in the test is deemed interesting and is further mutated by the fuzzer.

For example, the fuzzer initially starts with `input="welkrjerlkj"`. Running the program on this input results in covering **lines 2 and 9** in Figure 1. The fuzzer then saves this input and tries to create more inputs from it by mutating some of its characters. For example, it generates `"welkrjerlkj1111"`, `"telkrjeprgj"`, and `"felkrsbrlkj"`. Running the first two inputs does not achieve new coverage since the program again executes the same path **line 2, 9**. However, the third input changes the first character to `f`, which triggers the `then`-side of the if statement at (**lines 2, 4, 9**). New coverage is now achieved, and subsequent inputs will be mutated from this interesting one — the one that achieved the recent coverage, i.e., `felkrsbrlkj"`. This process continues until the correct input is constructed to explore the deepest part of the program — i.e., the `assert`-statement at **line 8**, at which point the assertion failure is triggered.

III. Motivation

Figure 2 and Figure 3 show two SV-COMP verification tasks that were unsolved by all verification tools participating in the 2025 Java Safety track, but were successfully solved by JQF-AFL. The first example in Figure 2 starts by defining two symbolic variables, m and n (**lines 12, 15**), and constraining them to be within the range 0–9999 (**lines 13–14, 16–17**). It then invokes the `addition` method, which increments the constant c while decrementing the variable n if $n > 0$, or decrements m otherwise (**lines 7–10**). The recursive function `addition` stops when $n == 0$ and fails when the constant $c \geq 150$ (**lines 3–6**). After the `addition` method terminates, the program checks that the returned value equals the sum of both m and n (**lines 19–20**). All verification tools fail to reach the assertion line 150. For example, Java Ranger runs out of memory, while JBMC, JDart runs times out in the 15 minutes budget, and finally, MLB fails to run the program. In particular, Java Ranger attempts to inline the recursive method deeply to explore the program state, but since each recursive step introduces two additional calls, the number of required

```
1.public class Main {
2.  public static int addition(int m, int n, int c){
3.    if (n == 0)
4.      return m;
5.    if (c >= 150)
6.      assert false;
7.    if (n > 0)
8.      return addition(m + 1, n - 1, ++c);
9.    else
10.     return addition(m - 1, n + 1, ++c);}
11.  public static void main(String[] args) {
12.    int m = Verifier.nondetInt();
13.    if (m < 0 || m >= 10000)
14.      return;
15.    int n = Verifier.nondetInt();
16.    if (n < 0 || n >= 10000)
17.      return;
18.    int c = 0;
19.    int result = addition(m, n, c);
20.    assert (result == m + n); }}
```

Fig. 2: Addition01: an SV-COMP example unsolved by all other tools, but successfully solved by JQF-AFL

```
1.public void benchmark10(double x, double y, double z) {
2. if (Math.sin(x * Math.cos(y * Math.sin(z)))
3.          > Math.cos(x * Math.sin(y * Math.cos(z))))
4.  assert false; }
```

Fig. 3: Coral10: an SV-COMP example unsolved by all other tools, but successfully solved by JQF-AFL

inlinings grows exponentially. To handle inputs as large as 200, Java Ranger would need to perform up to 2^{200} inlinings, which leads to memory exhaustion before completing the analysis.

On the other hand, JQF-AFL does not require any program analysis; it simply uses program coverage to identify interesting inputs and further mutates them to reach deeper parts of the program. Since the input space in this example is limited and the program is not very complex, JQF-AFL was able to reach the assertion within the 15 minute timeout.

Figure 3 shows another instance where fuzzing was able to reach the assertion, while none of the verification tools were able to do so. The program computes some trigonometric functions using the `sin` and `cos` functions over three free (symbolic) variables x, y, and z.

This particular example is not inherently difficult, as there is no complex branching—only a single conditional in the code. The real challenge comes from the need to support trigonometric functions as logical constraints over symbolic variables. In fact, none of the verification tools support these functions, which is why they fail to solve the task, i.e., they cannot reach the assertion. On the other hand, fuzzing does not require any special setup, translation, or treatment of Java functions; it simply tries different values for the free variables until it reaches the assertion line.

IV. Technique and Implementation

Figure 4 shows the general architecture of JQF-AFL for fuzzing Java programs, with our extension highlighted in gray. The process of fuzzing a Java program using JQF-AFL works

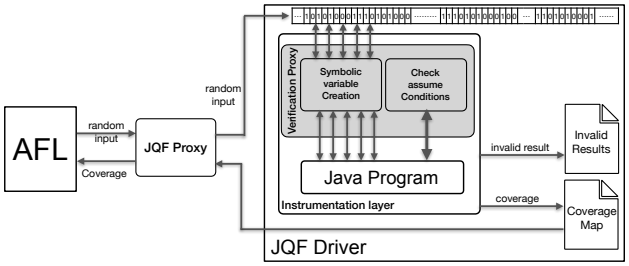

Fig. 4: Architecture of the extended JQF-AFL modified to run verification tasks. Our extension highlighted in gray

as follows. Initially, AFL passes a random input to the JQF proxy, which then forwards it to the JQF Driver. The JQF Driver runs this input on an instrumented version of the Java program. The instrumentation collects coverage information as the program executes. Additionally, the result of the execution is recorded. If the execution is successful, JQF skips storing the result; otherwise, the newly found result—often indicating a failure or bug—is added to the collection of invalid results. After execution completes, the JQF proxy sends the collected coverage map to AFL, which uses it to generate new random inputs. This process continues until a timeout is reached.

To support fuzzing verification tasks we have added a verification proxy which is responsible for two main functions: intercepting the creation of symbolic objects, and enforcing assumption checks.

A. Intercepting the Creation of Symbolic Objects

The idea here is that whenever the program-to-be-verified is about to request a new symbolic (fresh) variable, our process of symbolic variable creation intercepts the request and instead of creating a fresh symbolic variable, it creates the appropriate variable type based on the type of the object using the AFL-passed input. We support the creation of Java primitive types, including byte, char, int, boolean, long, float and double. In the implementation, the received input from AFL is intercepted as a stream of bytes, which depending on the object type requested by the program, our proxy creates the right object by reading and formatting the appropriate bytes from the input stream as per the Java language standards.

In addition to supporting primitive types, the object creation module can also create Java strings from the stream of bytes as UTF-8. Java String objects consist of Unicode characters, to support a variety of character sets and languages. For testing purposes, it is desirable to allow generated strings to contain any Unicode character, but to more frequently generate the more commonly-used characters such as those from ASCII. Therefore, we generate characters with a probability distribution matching the Unicode Transformation Format 8-bit encoding (UTF-8). UTF-8 encoding supports all valid Unicode code points using a *variable-width* encoding. The encoding can vary from using one byte to a maximum of four bytes. It uses specific bit-patterns to define valid UTF-8 sequences [7]. More precisely, a prefix of (0), (110 and 10), (1110, 10, and

Algorithm 1: Generating string for UTF-8 encoding with shifting and a geometric distribution

```
   input: input stream of bytes i
1  output: random utf-8 string s
2  data: single byte b
3  data: array of four bytes c, initially invalid utf-8 char
4  data: array of four bytes c'
5  data: number of bytes read n, initially 0
6  b = readNextByte(i)
7  while b > 25 do
8      repeat
9          while n < 4 do
10             b = readNextByte(i)
11             n = n+1
12             c[n] = b
13         end
14         c' = getUtf8(c)
15         if ¬isValidUtf8(c') then
16             c[0] = c[1]
17             c[1] = c[2]
18             c[2] = c[3]
19             c[3] = 0
20             n = n - 1
21         end
22     until ¬ isValidUtf8(c')
23     s = append(s, c')
24     b = readNextByte(i)
25 end
26 return s
```

10), and (11110, 10, 10, and 10) specifies the correct encoding for valid UTF-8 sequences of one-byte, two-bytes, three-bytes, and four-bytes, respectively. Thus, the expected probability of generating a valid UTF-8 sequence from a random stream of bytes is $(2^{31} + 2^{27} + 2^{24} + 2^{21}) \div 2^{32} \times 100 = 53\%$.

Algorithm 1 shows how we maximize the usage of a random stream of bytes to generate valid UTF-8 characters with the same probability, where the length of the random string is geometrically distributed with a mean of 10. The algorithm uses a shift-by-one approach to construct valid UTF-8 characters. More precisely, the algorithm takes an input stream of bytes i and produces a Java random string s. It begins by reading the next byte from the input stream into b (**line 6**). This value is used (**lines 7–24**) to simulate a geometric distribution with a mean of 10. This is enforced by excluding any value less than or equal to 25, since 26/256 (possible characters) $\approx 10\%$. Starting with an invalid UTF-8 character c', the loop between **lines 8–22** executes. It begins by filling the byte array c with four bytes from the input stream. It then uses the function $getUtf8(c)$ (omitted for simplicity) to check whether any of the prefixes in c result in a valid UTF-8 encoding, producing c'. If true, i.e., if c' holds a character that has a valid UTF-8 encoding, then that character is appended to the resulting string s (**line 23**), and a new byte b is read to attempt to increase the string's length (**line 24**). If c' does not contain a valid UTF-8 encoding, the contents of c are shifted by one byte to create a new four-byte candidate, reusing three bytes that were previously in c (**lines 16–20**). When no more bytes can be added to the output s, the constructed random string is returned (**line 26**).

B. Enforcing Assumption Checks

In verification, sometimes some initial constraints are enforced on input variables, called assumptions. These assumptions are usually added through a set of `assert` statements in Java. To support enforcing pre-condition constraints (assumptions) on input variables, we use a try-and-check approach. More precisely, we intercept any assumption on the symbolic input, then, we check whether the random object created from the AFL-passed input satisfies the assumption condition. If not, the Java program is terminated. The JQF proxy then requests a different input from AFL, and the entire process is repeated with a presumably different input. This continues until a valid object (i.e., one that satisfies the assumed condition) is created from the AFL-passed input. In our experiments, this approach works well—once a valid object is created, new coverage is collected, which leads AFL to recognize the input as interesting. As a result, it becomes the basis for generating new random inputs.

V. Evaluation

There are three research questions we were interested in answering in our evaluation: Can fuzzing solve verification tasks? What are the types of verifications tasks that fuzzing can have an advantage in? What is the performance of the fuzzer when used to solve verification problems? And finally, what are the strengths and limitations of fuzzing in the context of program verification?

To address these questions, we evaluate JQF-AFL [8] in comparison with state-of-the-art verification tools from the Software Verification Competition (SV-COMP). Since our primary interest lies in assessing the effectiveness of fuzzing as a complementary technique to formal verification–particularly in improving the reachability of property falsification–we restrict our experiments to the subset of unsafe verification tasks, where counterexamples are expected. For this set, fuzzers can be used to improve the reachability of counterexamples, and thus There are a total of 420 unsafe verification tasks in the latest version of SV-COMP (2025). We ran our extended version of JQF-AFL as well as other verification tools that exists in SV-COMP 2025, these included four tools.

- Java-Ranger [9]: a dynamic symbolic execution tool that tries to summarize pieces of branching code within the program to a logical formula to avoid the dynamic path-explosion problem.
- JBMC [10]: a bounded model checking tool for Java programs. It summarizes the entire code to be executed into a logical formula. While the tool avoids the dynamic explosion problem, the heavy lifting is handled by the

TABLE I: Number of correctly identified tasks vs incorrect and unknown ones. Tasks unknown in the first third of the budget (≤ 300 s) are considered premature unknowns

	JQF-AFL	Java-Ranger	JBMC	GDart	MLB
# of unsafe verification	277	287	321	383	390
# incorrect	0	1	0	0	5
# unknown	143	133	99	137	30
# premature unknown	25	124	90	130	23

TABLE II: Unique UNSAFE tasks that were found by JQF-AFL but missed by each tool respectively

	Java-Ranger	JBMC	GDart	MLB
unique tasks found by JQF-AFL	83	54	90	10
unique tasks found only by JQF-AFL	3			
unique tasks found by all except JQF-AFL	61			

solver used to find the satisifiablity of the generated constraint.

- GDart [11]: a modular framework designed to perform dynamic symbolic execution on JVM-based programs. It integrates three core components: a decision-making engine for symbolic exploration, a concolic execution module named SPouT, and JConstraints, an SMT-solving backend.
- MLB [12]: a dynamic symbolic execution tool that is driven by machine learning instead of relying on traditional solvers. In MLB, the feasibility problems of the path conditions are transformed into optimization problems, by minimizing some dissatisfaction degree. For the evaluation, we used MLB's version used in SV-COMP 2024, since the submitted version in 2025 seems to be broken.

We ran our experiments on a Dell Inc. Precision T3600 machine with 32 GB RAM running Ubuntu 22.04.4 LTS. We configured a time out of 15 minutes for all tools. Due to the inherent randomness of JQF-AFL, we repeated the experiment 10 times and report the statistical results over all runs.

A. Can fuzzing solve verification tasks?

Table I shows the number of unsafe tasks identified by each tool out of 420 total unsafe tasks. Although, JQF-AFL solved the least number of verification tasks, it was not too far off from Java-Ranger (277 vs. 287), the gold medal winner tool for the Java track in SV-COMP 2025. We also observe that JQF-AFL produces no incorrect results, similar to JBMC and GDart. In contrast, Java-Ranger yields a single incorrect result, and MLB incorrectly solves five. We attribute JQF-AFL's reliability to its simplicity: it executes programs directly, so any detected issue must exist. Java-ranger, however, relies on task summarization, which is error-prone, while MLB uses a solving technique that is incomplete because their search process is unsound.

TABLE III: Number of UNSAFE tasks per SV-COMP category. Categories highlighted in gray indicate $> 75\%$ verified by JQF-AFL. Here JR refers to Java-Ranger

	count	JQF-AFL	JR	JBMC	GDart	MLB
algorithms	22	12	20	19	16	19
float-nonlinear	74	57	5	15	4	68
java-ranger-reg.	10	10	8	5	6	7
jayhorn-rec.	9	3	6	8	4	8
jbmc-reg.	87	78	64	81	63	84
jdart-reg.	13	13	8	11	11	12
jpf-reg.	52	52	47	51	49	52
juliet-java	6	0	0	0	0	3
mine-pump	56	56	56	56	52	56
securibench	91	0	73	75	78	81
Total	420	281	287	321	283	390

Finally, we observed that JQF-AFL reported 143 unknown tasks, compared with 137 for GDart, 133 for Java Ranger, 99 for JBMC, and 30 for MLB. Notably, only 25 of JQF-AFL's unknowns terminated prematurely (within the first third of the time budget), whereas Java Ranger, JBMC, GDart, and MLB had 124, 90, 130, and 23 premature unknowns, respectively. For verification tools, the high ratio of premature to total unknowns largely stems from unsupported language features and sometimes tool bugs. In contrast, JQF-AFL's low premature rate (25/139) highlights its scalability and absence of verification tool's limitations; its premature cases arose instead from other runtime failures (e.g., `InputMismatchException`, `StringIndexOutOfBoundsException`) rather than the expected `AssertionError` indicating a property violation.

To assess JQF-AFL's unique performance, we analyzed the unsafe tasks it identified but were missed by other tools. Table II shows that, among the 281 unsafe tasks found by JQF-AFL, 83 were not found by Java-Ranger, 54 by JBMC, 90 by GDart, and 10 by MLB. Notably, 3 of these tasks were not invalidated by all the tools combined. Looking at the reverse case–tasks solved by all tools but missed by JQF-AFL–we found 61 such tasks. Among the 61 tasks, 14 of them JQF-AFL stopped prematurely because it detected a different failure/exception (e.g., `InputMismatchException`, `StringIndexOutOfBoundsException`) than the assertion error (`java.lang.AssertionError`) expected from property violation. Also, in five cases, JQF-AFL ran out of memory. The remaining tasks involved difficult string-construction problems, which are generally better handled by verification tools than fuzzers.

B. What are the types of verifications tasks that fuzzing can have an advantage in?

To further evaluate the tasks were JQF-AFL was better at, we analyzed the different categories of tasks within the SV-COMP. Table III show the different category of tasks with the SV-COMP. In total there are 10 different categories with 420 unsafe task. The table shows the total number of the tasks within each category and the number of unsafe tasks found by each of the tools. JQF-AFL reached over 75% of the unsafe tasks in 6 of the 10 categories (float-nonlinear, java-reg., jbmc-

TABLE IV: Runtime performance for each tool, all figures is computed from CPU time, in seconds. The last row shows the number of tasks that took over 800 seconds (for implementation reasons, the timeout threshold varies by tool between approximately 800 and 900 seconds).

	JQF-AFL	Java-Ranger	JBMC	GDart	MLB
average	244.77	29.50	20.97	27.68	27.57
median	9.99	7.19	2.44	10.98	5.07
stdev	373.35	124.30	111.95	108.50	123.33
max	876.59	803.63	878.65	901.02	900.92
min	8.85	2.54	1.04	7.06	4.20
unsafe-avg	20.17	17.46	2.94	13	11.79
$\geq 800s$	111	5	6	6	8

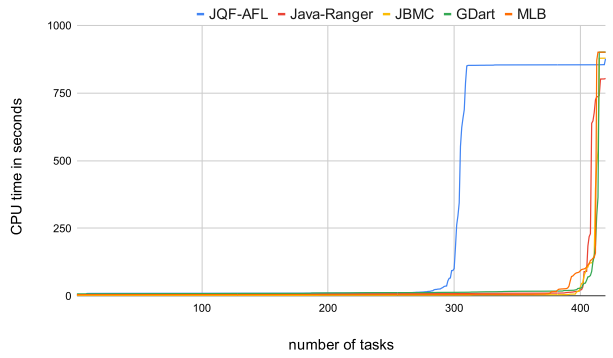

Fig. 5: Execution of each tool, presented in CPU time.

reg., jdart-reg., jpf-reg., mine-pump). Most notably, it found 57 of 74 faults in non-linear problems, while jr (5/74), JBM (15/74), and GDart (4/74) struggled. This reflects a limitation of complete verification methods, which face undecidability and solver challenges with non-linear constraints. By contrast, fuzzing executes programs with concrete random inputs and gathers coverage, making it more likely to expose faults in such cases. Both MLB and JQF-AFL are incomplete tools, but MLB's heuristic-driven search was more effective than JQF-AFL's random strategy on these benchmarks.

On the other hand, JQF-AFL performed poorly on `securibench`, with an average success rate of zero across the 91 tasks. Upon analysis, we found that the verification tasks in this benchmark category depend on matching specific string patterns, such as `"<bad/>"`. While a fuzzer could eventually discover such patterns, it may require more time than the 15 minutes allocated for the experiment, and it might also benefit from a dictionary of likely useful strings. In contrast, other verification tools, if they support string operations, can detect these patterns more efficiently by encoding them as constraints and solving them symbolically.

C. What is the performance of the fuzzer when used to solve verification problems?

Figure 5 shows the runtime performance of all tools. JQF-AFL was the slowest, but it completed about 3/4 of the unsafe tasks within 100 seconds. For the remaining $\sim$1/4 ($\approx$ 100 tasks), it exhausted the full 15-minute budget. As table IV shows, its number of `unknown` results was comparable to GDart and Java-Ranger. However, table I reveals that JQF-AFL used the full budget on 111 of 143 tasks, unlike Java-Ranger, JBMC, GDart, and MLB, which only timed out ($\geq$ 800s) on 5, 6, 6, and 8 tasks, respectively.

Table IV shows the performance statistics for all five tools. JQF-AFL has the highest standard deviation ($\approx$ 373s) due to alternating between early termination and full-time usage. Java-Ranger, JBMC, and GDart have similar averages (between 20-30 seconds), reflecting their common use of SAT solvers. MLB is often faster when it quickly finds a verdict, as it can save time wasted on querying SMT solvers. On the other hand, JQF-AFL tends to exhaust the full budget to maximize chances of reaching unsafe code.

VI. Discussion

Fuzzing proved most effective when verification tools struggled with theoretical limits, such as non-linear constraints. It also scales better, since it requires no special encoding. For instance, our experiments showed missing symbolic support for methods like `String.regionMatches()` in both JBMC and Java-Ranger, and `Math.toRadians()` in JBMC.

On the other hand, fuzzers are not well suited for verification tasks that require certain assumptions to be satisfied before execution. In our evaluation, we allowed the fuzzer to fail if some predetermined assumption was not met, and regenerate new ones. However, this can significantly affect the fuzzer's performance when the chance of randomly generating a valid input is low. One approach to improve the fuzzer's effectiveness is to allow it to use special inputs, possibly generated by a verification tool. In general, other works have explored analyzing programs using a combination of verification and fuzzing [13], [14], and have shown promising results by leveraging the strengths of both approaches.

VII. Related Work

Many works have attempted to develop both bounded and unbounded modeling techniques [2], [9]–[12], [15]–[31] to verify various program properties, such as reachability [18], [19], [24], memory safety [18], [32], [33], termination [29], [33], overflows [33]–[35], and concurrency [20], [21], [36]. Some of these techniques rely purely on static execution [10], [16], while others combine static and dynamic execution [9], [15], [26], [27], [37], [38], with the latter often offering better scalability.

On the other hand, fuzzing is a well-known random testing technique with many variations. In coverage-guided fuzzing (CGF) [5], [39]–[44] the fuzzer uses random testing with lightweight instrumentation of the program under test for coverage computation. Using the coverage information, fuzzing tools can then make informed decisions about the choice of interesting input likely to achieve new coverage if mutated. CGF revealed many bugs in widely used/tested programs, such as in Clang, OpenSSH, JavaScriptCore, LibreOffice, Python,

SQLite, Google closure-compiler, JDK, Mozilla, and BCEL to mention a few [5], [6], [45].

However, as CGF lacks information about the input structure, its effectiveness can be negatively affected. Grammar-based fuzzers such as CSmith [46], jsfunfuzz [47], and Grammarinator [48] use a declarative form of the input that defines input's grammar. Generator-based fuzzers (GBF) utilize programmatically defined generators to create structured inputs.

Some work proposes hybrid approaches that combine verification and fuzzing [13], [14], [49]. The advantage is that symbolic execution can turn input search into constraints; solving them produces inputs that reliably hit targeted paths.

This paper evaluates coverage-guided fuzzing for verification. Our results show it outperforms other tools in some cases, and we expect combining it with traditional verification could yield even better outcomes.

VIII. Future Work

Our findings suggest that fuzzers can be significantly more effective when guided toward input values that are semantically meaningful for the verification task. One promising direction is to explore hybrid approaches, where symbolic analysis or static analysis is used to extract input constraints or likely input shapes, which are then used to seed or guide the fuzzer. Additionally, maintaining a dynamic dictionary of program-specific constants, especially string patterns observed during execution or analysis (e.g., tags, error codes), could help the fuzzer target deeper or otherwise unreachable branches. Investigating how to integrate such dynamic knowledge sources while preserving fuzzing efficiency and scalability remains an interesting and open problem.

IX. Conclusion

In this paper, we showed how we used JQF-AFL, a Java fuzzer, to verify unsafe verification tasks from SV-COMP 2025. To evaluate the modified JQF-AFL extension, we tested it on 420 verification tasks from SV-COMP 2025 that exhibit unsafe properties. Our results show that the fuzzer was able to reach the unsafe statement in 281 of these tasks, including 3 that were not found by any of the verification tools. From our findings, we observed that the fuzzer can outperform traditional verification tools in tasks where summarizing the program into logical constraints is either difficult or simply, not supported. Additionally, because the fuzzer explores one path at a time, it avoids the path and state explosion that often limits verification techniques. Overall, our comprehensive evaluation suggests that incorporating fuzzing into the verification process is an effective strategy, especially in cases where verification tools face limitations.

References

[1] AWS, "Automated reasoning," https://www.amazon.science/research-areas/automated-reasoning, [Online; accessed 6-july-2025].

[2] S. Priya, Y. Su, Y. Bao, X. Zhou, Y. Vizel, and A. Gurfinkel, "Bounded model checking for llvm," in *2022 Formal Methods in Computer-Aided Design (FMCAD)*, 2022, pp. 214–224.

[3] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, and T. Xu, "Anvil: Verifying liveness of cluster management controllers," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 649–666. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[4] D. M. Groß, "Towards trustworthy ai: Formal verification in machine learning," Ph.D. dissertation, Sl: sn, 2024.

[5] M. Z. 2014, "American fuzzy lop (afl)," https://lcamtuf.coredump.cx/afl/.

[6] R. Padhye, "Jqf github," https://github.com/rohanpadhye/JQF.

[7] F. Yergeau, "RFC 3629: UTF-8, a transformation format of ISO 10646," https://www.rfc-editor.org/rfc/rfc3629, [Online; accessed 6-july-2025].

[8] S. Hussein, "AFL-JQF," https://github.com/sohah/JQF-AFL, accessed: Oct 5, 2025.

[9] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser, "Java ranger: statically summarizing regions for efficient symbolic execution of java," 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, p. 123–134. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1145/3368089.3409734

[10] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "JBMC: A bounded model checking tool for verifying Java bytecode," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 10981. Springer, 2018, pp. 183–190.

[11] M. Mues and F. Howar, "Gdart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 435–439.

[12] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li, "Symbolic execution of complex program driven by machine learning based constraint solving," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 554–559.

[13] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[14] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *SIGPLAN Not.*, vol. 43, no. 6, p. 206–215, jun 2008.

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

[16] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[17] L. Bu, Z. Xie, L. Lyu, Y. Li, X. Guo, J. Zhao, and X. Li, "Brick: Path enumeration based bounded reachability checking of c program (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 408–412. [Online]. Available: https://doi.org/10.1007/978-3-030-99527-0_22

[18] D. Beyer and M. E. Keremoglu, "Cpachecker: a tool for configurable software verification," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 184–190.

[19] P.-C. Chien and N.-Z. Lee, "Cpv: A circuit-based program verifier," in *Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 365–370. [Online]. Available: https://doi.org/10.1007/978-3-031-57256-2_22

[20] H. Ponce-de León, F. Furbach, K. Heljanko, and R. Meyer, "Portability analysis for weak memory models porthos: One tool for all models," in *Static Analysis*, F. Ranzato, Ed. Cham: Springer International Publishing, 2017, pp. 299–320.

[21] F. He, Z. Sun, and H. Fan, "Deagle: An smt-based verifier for multi-threaded programs (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 424–428.

[22] L. Bajczi, D. Szekeres, M. Mondok, Z. Ádám, M. Somorjai, C. Telbisz, M. Dobos-Kovács, and V. Molnár, "Emergentheta: Verification beyond abstraction refinement (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 371–375. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1007/978-3-031-57256-2_23

[23] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 331–340. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1145/1985793.1985839

[24] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, p. 957–974, Jul. 2012. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1109/TSE.2011.59

[25] C. Telbisz, L. Bajczi, D. Szekeres, and A. V¨or¨os, "Theta: Various approaches for concurrent program verification (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2025, p. 260–265. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1007/978-3-031-90660-2_22

[26] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: Symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 179–180.

[27] C. Artho, P. Parízek, D. Qu, V. Galgali, and P. L. Yi, "Jpf: From 2003 to 2023," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 3–22.

[28] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The jkind model checker," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10982 LNCS, pp. 20–27, jul 2018.

[29] J. Hensel, C. Mensendiek, and J. Giesl, "Aprove: Non-termination witnesses for c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 403–407.

[30] M. Chalupa and T. A. Henzinger, "Bubaak: Runtime monitoring of program verifiers: (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 535–540. [Online]. Available: https://doi.org/10.1007/978-3-031-30820-8_32

[31] H. Mousavi, "Jayhorn:java and horn clauses," {https://github.com/jayhorn/jayhornl}.

[32] M. Jonáš, K. Kumor, J. Novák, J. Sedláček, M. Trtík, L. Zaoral, P. Ayaziová, and J. Strejček, *Symbiotic 10: Lazy Memory Initialization and Compact Symbolic Execution: (Competition Contribution)*, 04 2024, pp. 406–411.

[33] M. Heizmann, M. Barth, D. Dietsch, L. Fichtner, J. Hoenicke, D. Klumpp, M. Naouar, T. Schindler, F. Sch¨ussele, and A. Podelski, "Ultimate automizer and the commuhash normal form," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Sankaranarayanan and N. Sharygina, Eds. Cham: Springer Nature Switzerland, 2023, pp. 577–581.

[34] R. Xu, F. He, and B.-Y. Wang, "Interval counterexamples for loop invariant learning," 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, p. 111–122. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1145/3368089.3409752

[35] A. Nutz, D. Dietsch, M. M. Mohamed, and A. Podelski, "Ultimate kojak with memory safety checks," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Berlin, Heidelberg: Springer-Verlag, 2015, p. 458–460. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1007/978-3-662-46681-0_44

[36] A. Farzan, D. Klumpp, and A. Podelski, "Sound sequentialization for concurrent program verification," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 506–521. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1145/3519939.3523727

[37] N. Loose, F. M¨achtle, F. Sieck, and T. Eisenbarth, "Swat: Modular dynamic symbolic execution for java applications using dynamic instrumentation (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 399–405. [Online]. Available: https://doi-org.ezp2.lib.umn.edu/10.1007/978-3-031-57256-2_28

[38] K. Luckow, M. Dimjasevic, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamaric, and V. Raman, "Jdart: A dynamic symbolic analysis framework," in *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, M. Chechik and J.-F. Raskin, Eds., vol. 9636. Springer, 2016, pp. 442–459.

[39] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[40] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[41] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[42] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4901–4911.

[43] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 787–802.

[44] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[45] LibFuzzer, "Libfuzzer trophies," https://llvm.org/docs/LibFuzzer.html#trophies.

[46] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[47] W. Syndder and M. Shaver, "jsfunfuzz," https://github.com/MozillaSecurity/funfuzz?tab=readme-ov-file.

[48] R. Hodován, A. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 45–48. [Online]. Available: https://doi.org/10.1145/3278186.3278193

[49] J. Kukucka, L. Pina, P. Ammann, and J. Bell, "Confetti: Amplifying concolic guidance for fuzzers," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 438–450.