Toward Using Fuzzers and Lightweight Specifications to Reveal Semantic Bugs

Amirfarhad Nilizadeh[*], **Gary T. Leavens**[*], **and David R. Cok**[†]
[*]University of Central Florida, USA
[†]Safer Software Consulting, LLC, USA

ABSTRACT Although fuzzers have been successful in revealing semantic bugs that lead to crashes, they do not reveal semantic bugs that do not lead to crashes. Furthermore, the inputs that lead to crashes may be invalid and invalid inputs do not reveal semantic bugs at all, since they are outside the program's intended input domain. On the other hand, runtime assertion checking (RAC) may be used for revealing semantic bugs, although it needs input test data that can trigger these bugs.
In this idea paper, we propose the idea of combining different kinds of fuzzing tools and RAC in a complementary manner to leverage their benefits and overcome these problems, along with a preliminary study. That is, a fuzzing tool will generate an input test, and a RAC tool will make sure that the generated inputs are valid and check the results for semantic bugs.

KEYWORDS Software Testing, Fuzzing, Runtime Assertion Checking, Lightweight Specifications

1. Introduction and Motivation

Detecting software bugs is time-consuming and challenging. Many techniques have been introduced for discovering input tests that can reveal bugs early in a system's development. Generating tests that can reveal bugs would also help other software research domains like automatic bug localization (Wong et al. 2016), program synthesis (Alur et al. 2018; Peleg & Polikarpova 2020; A. Nilizadeh, Calvo, et al. 2022) and automated program repair (Gazzola et al. 2018; Goues et al. 2019; A. Nilizadeh & Leavens 2022; A. Nilizadeh et al. 2024).

Two practical approaches for automatically discovering bugs in programs are fuzzing (Liang et al. 2018; Li et al. 2018; Boehme et al. 2021) and runtime assertion checking (RAC) (Clarke & Rosenblum 2006; Leucker & Schallhart 2009; Kosmatov et al. 2020; Taleb et al. 2021). Used alone, fuzzing can only reveal program crashes as opposed to semantic bugs, while RAC can reveal semantic bugs but needs valid inputs. Our idea is to combine these two approaches to reveal semantic bugs more effectively automatically; that is, the combination should be more *effective* in that it reveals more bugs in a given amount of time.

Fuzzing is the process of running a program repeatedly with randomly generated inputs to reveal vulnerabilities or security violations by monitoring crashes. This process is repeated until a timeout is reached or fuzzing is stopped. Fuzzing is used routinely by well-known companies like Google (Chang et al. 2017) and Microsoft (Godefroid et al. 2012; Microsoft Corp. 2015). Fuzzers can be classified into three groups: black-box, gray-box, and white-box (Manès et al. 2019). In *black-box fuzzing*, the source code and internal behavior of the program under test (PUT) are not accessible. Thus, generating inputs in a black-box fuzzer is entirely random. In *white-box fuzzing*, the code is available, and fuzzing is not necessarily random, as the fuzzer can explore the PUT's state space; an example is SAGE (Godefroid et al. 2012), which uses concolic execution to generate inputs that follow specific paths. However, white-box fuzzing needs program code to work, and its use of static analysis can be slower than the random input generation in black-box fuzzers. *Gray-box fuzzing* can see some internal information of the PUT, but not all of it. Gray-box fuzzing uses lightweight static analysis and/or gathers dynamic information from the PUT's execution, such as code coverage. Gray-box fuzzing uses randomly generated inputs; however, it can use some internal information to generate better test inputs.

JOT reference format:
Amirfarhad Nilizadeh, Gary T. Leavens, and David R. Cok. *Toward Using Fuzzers and Lightweight Specifications : to Reveal Semantic Bugs* . Journal of Object Technology. Vol. 23, No. 2, 2024. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2024.23.2.a3

Fuzzers are designed to reveal bugs that lead to a program crash; fuzzers cannot detect semantic bugs that do not cause the PUT to crash, although it should be noted that assertion violations (i.e., assertions that evaluate to false) typically lead to a crash. Thus, if the program has assertions, fuzz testing can be used to check for assertions that are violated. On the other hand, some crashes are not interesting, because: (1) the inputs used may be invalid (i.e., outside of the program's input domain) or (2) the crash may be intentional (e.g., a program may abort or throw an exception to signal an invalid input).

RAC checks a lightweight formal specification while running a test by evaluating assertions; when RAC detects a specification violation, it reveals a semantic bug. However, like software testing, RAC is incomplete. RAC can only reveal a bug when the input test data covers a path that violates an assertion, and RAC cannot guarantee that a program with no violations is correct for all possible inputs. Thus, the effectiveness of RAC for revealing bugs is dependent on the input test data. However, when using RAC together with a specification language, such as JML (Leavens et al. 2006, 2008, 2022) or Eiffel (Meyer 1997), one does not provide the expected output for each input but instead a lightweight specification for each method, consisting of a precondition and a postcondition. A *Precondition* is an assertion that describes what input states are valid, and a *postcondition* describes the correct relation between input and output states.

If one has a formal specification for a program, then why use input tests and RAC instead of static verification? The problem is that writing a formal specification that allows successful static verification is time-consuming, and many commonly used libraries do not have such formal specifications. When using such a library, the lack of formal specifications for the library's methods prevents verification of code that calls them.

By contrast, it is easier to write lightweight formal specifications to be used in RAC, because details like loop invariants can be omitted, and one needs only to write pre- and post-condition specifications for methods called (and perhaps some assertions that describe assumed properties of data structures (Hoare 1972)). Since RAC is not attempting to prove programs correct, its lightweight specifications may not suffice for static verification; however, they can still reveal semantic bugs.

The main idea of this paper is to discuss the prospect of extending fuzzing to reveal semantic bugs by combining fuzzers with RAC using lightweight specifications. Some potential benefits are: (1) unit tests need not be written, (2) running the PUT with valid input data by using RAC and the PUT's precondition and discarding invalid inputs[1], (3) revealing semantic bugs by observing contract violations, and (4) sending feedback to the fuzzer to help it provide better coverage and/or reveal bugs.

2. Background

In this section, we discuss the background of coverage-guided fuzzer tools and RAC, along with an introduction to the specification language employed in our preliminary study.

[1] A PUT's precondition gives constraints on its input domain, which say what top-level inputs (and states) are allowed. However, if an internal call within the PUT causes a precondition failure, that is indicative of a program bug.

2.1. Coverage-Guided Fuzzers

Recall that fuzzing is the process of generating random inputs to reveal bugs that lead to program crashes. However, modern fuzzers do not work entirely randomly.

A *coverage-guided fuzzer* (CGF) uses feedback from program execution to enhance code coverage. This is achieved by diversifying the generated input tests, as exemplified by tools such as AFL (Zalewski 2014), LibFuzze (Serebryany 2016), VUzzer (Rawat et al. 2017) and Tensorfuzz (Odena et al. 2019). These tools all follow the same general steps to test a program, *PUT*. First, they have a pool of *good inputs*, namely those that do not crash *PUT* or appear to send it into an infinite loop. Then, they select a good input for the mutation process to generate a new test. They then execute *PUT* with the newly generated input. They monitor *PUT* to see if the new input increases code coverage or discovers a crash (or apparent infinite loop) during execution. The fuzzer reports the problems it discovers; furthermore, if the newly generated input increases code coverage, then it is added to the pool of good inputs. This process is repeated until a timeout is reached or fuzzing is stopped.

For using a CGF, a tester needs to provide a driver (usually written manually) to translate the input data generated by the fuzzer into *PUT*'s argument types. Writing a driver increases the probability of generating valid inputs for complex arguments to *PUT* by generating arguments of the expected types that should be valid. However, there is no guarantee that a driver will only call *PUT* with valid inputs as the driver may generate data that does not satisfy *PUT*'s precondition.

Some CGF tools allow the user to provide feedback on a second metric (in addition to code coverage). Such a fuzzer saves inputs that increase this user-defined metric; these fuzzers also save inputs that increase code coverage or reveal a crash. For example, DifFuzz (S. Nilizadeh et al. 2019) saves an input test that increases the time or space used by the PUT while searching for side-channel vulnerabilities in a (cryptographic) program.

When using a CGF, the tester must provide at least one good input to seed the pool of good inputs. Research shows that good initial input(s) will help CGF tools to generate good inputs in a shorter time (Yue et al. 2019). The idea of a *hybrid fuzzer* is to provide high-quality initial test data by using symbolic execution. Research shows the effectiveness of symbolic execution techniques for generating such input test data (Clarke 1976; Cadar et al. 2008; Braione et al. 2017). In hybrid fuzzing, symbolic execution generates good inputs and adds these to the pool of good test inputs to help guide fuzzers to generate better input tests; examples include Mayhem (Cha et al. 2012), Driller (Stephens et al. 2016), Badger (Noller et al. 2018) and HyDiff (Noller et al. 2020).

2.2. Runtime Assertion Checking

Runtime assertion checking (RAC) has intermediate benefits for program correctness—between formal methods and software testing. RAC has intermediate effectiveness because it checks formal specifications by running input tests to find violations of specified assertions (Clarke & Rosenblum 2006; Kosmatov et al. 2020; Filliâtre & Pascutto 2021). Thus RAC cannot guarantee the correctness of a program, but using assertions is effective for revealing semantic bugs. RAC can also provide information about bug locations, even for program bugs that do not lead to crashes or failures. However, when using RAC, one must provide preconditions and postconditions for the PUT's methods (or functions). In addition, inline assertions can be added to the PUT to check other program points, which is helpful for fault localization purposes if any such assertion is violated.

Observing assertion violations shows the existence of a bug only if the input data passed to the PUT's entry method[2] were valid (Peters & Parnas 1994; Cheon & Leavens 2002); that is, these input data must satisfy *PUT*'s precondition, otherise a contract violation encountered while running *PUT* has no meaning. Similarly, if a RAC tool does not show any postcondition violation, it only means that *PUT* works correctly on the provided input data. It may still be that there is a bug in the program that these inputs do not reveal or that RAC the specifications may be incomplete or not amenable to RAC; thus, running RAC without finding an assertion violation does not guarantee correctness for all possible inputs.

2.3. Java Modeling Language

The Java Modeling Language (JML) (Leavens et al. 2006, 2008, 2022) is a well-known Hoare-style formal behavioral interface specification language specifically designed for Java. Serving as a deductive specification language, JML describes the expected behavior of Java methods, classes, and interfaces using preconditions and postconditions. Furthermore, it offers annotations that allow for the crafting of inline assertions and loop invariants. JML specifications are integrated into Java code as formatted comments. These can be placed in-line after a comment starting with `//@` or between the special comment tags `/*@` and `*/`. Thus Java interprets JML annotations as comments.

Preconditions are assertions that must be satisfied before invoking a method or constructor. Conversely, postconditions are assertions that need to be fulfilled at the termination point of a method or constructor. Inline assertions amidst the code's statements must be satisfied when the execution reaches them. Additionally, loop invariants, a type of inline assertion, must be maintained at the beginning of every loop iteration.

A variety of tools, like OpenJML (Chalin et al. 2006; Cok 2010, 2011, 2021), and KeY (Ahrendt et al. 2014, 2016), leverage JML. While KeY exclusively supports static verification, OpenJML offers both static verification and RAC functionalities. OpenJML's RAC merely requires method specifications for the method under test; however, it can use inline assertions to monitor test executions for assertion violations.

Figure 1 shows an example of a Java class named `PrimeCheck` from the Java+JML dataset.[3] This class contains two methods, each annotated with JML pre- and postconditions.

[2] The *entry method* of a program is the method called to start the program's execution.

[3] The programs in our Java+JML dataset have complete specifications; that is each has specifications sufficient to allow formal verification. However, for this work, we only used method pre- and postconditions, excluding inner assertions and loop invariants; although those could also be checked by RAC, our experiments did not use inner assertions and loop invariants.

```java
class PrimeCheck {
   /*@    requires d != 0;
     @    ensures \result == n%d; @*/
    public /*@ pure @*/ static int
      div(int n, int d) { return n%d; }

   /*@ requires 1 < a;
     @ ensures \result <==> (\forall int k;
       1 < k && k <= a/2; div(a, k) != 0); @*/
   public boolean isPrime(int a) {
     int i = 2;
     int mid = a/2;
     while (i <= mid) {
      if (div(a,i) == 0) { return false; }
      i++;
     }
     return true;
   }
}
```

Figure 1 Java Code Example using pre- and postconditions in JML.

The first method, `div`, takes two parameters: the dividend (`n`) and divisor (`d`), and returns the remainder. The precondition (following `requires`) stipulates that `d` must not be zero. The postcondition (following `ensures`), on the other hand, checks that the return value matches `n%d`. The `pure` clause says that this method has no side effects, and allows it to be used in assertions.

The second method, `isPrime`, determines if the input value is prime. The precondition specifies that the input (`a`) must be strictly greater than 1. The postcondition says that the result is true just when the argument (`a`) is not divisible by any integer greater than one and less than or equal to `a/2`, so `a` is prime. In the postcondition, `\result` denotes the method's return value, `<==>` means "if and only if", and `\forall` is used for universal quantification. In a universally quantified expression, the type of the quantified variable (`k` in this case) is declared, and each value of that variable that satisfies the range condition (in this case `1 < k && k <= a/2`) must satisfy the condition given in the body of the quantifier (i.e., `div(a,k) != 0`).

With RAC, a contract violation happens if an input argument fails the precondition or the postcondition. If an input argument does not satisfy the precondition, the input is considered invalid, meaning an issue with the input data. When the input is valid, then a failure in a postcondition or an internal assertion indicates a program bug. Recognizing these differences, RAC can be effectively used to distinguish between input data errors and internal bugs, improving the bug detection process.

3. Approach

This section discusses our plans to solve the problem of revealing semantic bugs. Then, we discuss the application of RAC to different kinds of fuzzers, which involves the advantages, disadvantages, and potential areas of improvement when merging RAC with these different kinds of fuzzing tools.

3.1. A High-Level Solution

Our main idea is to combine fuzzing and RAC, using RAC to reveal semantic bugs and using a fuzzer to generate input data for RAC. Recall that a fuzzer can only reveal semantic bugs that lead to a crash, and RAC can only reveal semantic bugs if it is provided with suitable input test data. Notice that a RAC only needs input test data and not a test suite with inputs and their expected outputs. Thus, our idea is to use a fuzzer and a RAC in tandem to leverage their benefits and overcome their weaknesses to reveal semantic bugs effectively.

The inputs in our approach are a program, *PUT*, and its lightweight formal specification S in the Hoare style (Hoare 1969; Meyer 1997; Apt & Olderog 2019); that is, for the PUT's entry method, there is a contract consisting of a precondition and a postcondition; optionally, one can also specify other methods (and it is often convenient to add some invariants that describe data structures (Hoare 1972)). Then, a fuzzer will generate input tests, and a RAC tool will check the inputs to *PUT* and only pass along those that are valid, and then it will check S while running *PUT* to reveal contract violations. The output of this combination is two pools of test data: (a) those that pass S and (b) those that reveal semantic bugs (i.e., that violate the contracts in S). In other words, the high-level solution is to develop a fuzzing tool for running *PUT* with only valid input and detecting bugs if a contract is violated. Recall that generating valid inputs refers to providing inputs that satisfy the constraints of top-level inputs for *PUT*'s entry method. If there are other specifications, for example for other methods, then a failure in one of these indicates a program bug.

3.2. Using RAC to Check for Valid Inputs

The first step for combining a fuzzer with RAC is generating valid input data. When the input argument is invalid based on the input domain (precondition), RAC cannot detect potential semantic bugs in the program. Thus, one of the main conditions for using a RAC with a fuzzer is having a (lightweight) specification for the program's entry method. This precondition allows the RAC tool to check that the input is valid before running the program. Checking this precondition will guarantee that the program will only execute with valid inputs. In this case, after generating input test data with a fuzzer, the RAC will check the PUT's entry method precondition. If the generated input test does not satisfy the entry method's precondition, then the PUT will not be run; instead, the process will be repeated with the fuzzer generating new input data. Only when the input data satisfies the entry method's precondition will the entire PUT be run.

3.3. Benefits With Each Kind of Fuzzer

A variety of different fuzzing techniques can be used with RAC. In the following, we will discuss the benefits and limitations of each kind of fuzzing tool.

3.3.1. Black-Box Fuzzing In black-box fuzzing, a fuzzer generates input tests randomly, and the fuzzer has no information about how the PUT is implemented. Thus, having a lightweight specification that is suitable for black-box fuzzing only requires pre- and postconditions for the PUT's entry method; detailed specifications for other methods, loops and data structures are not needed. Since black-box fuzzers generate input data randomly, they may generate invalid input data (i.e., data that do not satisfy the PUT's entry precondition). Invalid inputs are more likely to be a problem if the PUT has more complex types of input data or more constrained input domains. Thus, if the input data are not checked for validity, then black-box fuzzing may run the PUT with much invalid input data, which is both time-consuming[4] and a waste of time, as such inputs cannot reveal semantic bugs.

A RAC tool can help to solve some of these limitations of black-box fuzzing. First, checking preconditions will ignore invalid input data and save time by only running a PUT with valid inputs. Furthermore, by running a PUT with valid input data, a RAC tool can reveal semantic bugs by checking for postcondition violations; since then, any postcondition violation shows the existence of a semantic bug. In sum, in combination with black-box fuzzing, the RAC tool can ignore invalid input data by checking the PUT's overall precondition and revealing semantic bugs using the PUT's overall postcondition (and any other contracts that are available).

3.3.2. CGF In coverage-guided fuzzing (CGF), the fuzzer can access some information in the PUT. Consequently, when integrated with RAC, a CGF tool retains all the benefits associated with black-box fuzzing. Additionally, by combining CGF tool with RAC, such a tool leverages its insight into the PUT to: (1) improve bug localization and (2) find semantic bugs more efficiently, as a CGF tool generates inputs by a process that attempts to maximize code coverage, instead of being completely random as in the black-box approach.

Furthermore, a RAC can improve bug localization if it is supplied with assertions internal to the PUT (e.g., contracts for some internally used methods). In this case, assertion violations can help to detect bugs during the PUT's execution instead of at the end of the PUT's execution (Barner et al. 2005).

Finally, the combination of a RAC with a CGF can ensure that the PUT is only run with valid input tests. This will be more efficient than using random inputs because the pool of good test data that a CGF mutates to generate new tests will only contain valid inputs (since the RAC checks the overall precondition of the PUT); such valid inputs are more likely to be mutated into other valid inputs and these can be tested more efficiently than invalid inputs.

3.3.3. Hybrid Fuzzing The distinction between guided and hybrid fuzzing lies in the input generation. In hybrid fuzzing, inputs are derived both from the fuzzer and symbolic (or concolic) execution (Jiang et al. 2023), with the latter being responsible for producing the initial inputs. Thus the symbolic execution

[4] Invalid inputs may consume much time because such inputs can lead to semantic errors, crashes, and infinite loops.

engine may seed the pool of quality inputs with inputs that cover many of the PUT's branches. Afterward, the fuzzer can mutate these inputs to create new ones. When combined with RAC, each input is checked for validity. Therefore, when hybrid fuzzing works with RAC, it harnesses the advantages of both CGF and RAC tools. This synergy can further enhance the efficiency of generating valid test data, especially since the initial seed inputs are also valid.

3.3.4. CGF with User-Defined Rewards

In some CGF tools like Kelinci-WCA (Noller et al. 2018; S. Nilizadeh et al. 2019), a tester can define a measure[5] that the fuzzer attempts to increase while it also attempts to increase code coverage. The main advantage of combining RAC with such a fuzzer is that the runtime assertion checker can send direct feedback signals to the fuzzer, allowing the fuzzer to find valid input test data that reveal semantic bugs more efficiently.

A combined RAC and fuzzer tool can prioritize revealing semantic bugs by having RAC directly reward the fuzzer when its input test data reveals a semantic bug. Alternatively, such a combined tool can prioritize finding valid inputs by rewarding the fuzzer when it generates valid input test data[6]. Fig. 2 shows a high-level diagram of this approach. In this diagram, first, a CGF tool generates input data. Then, a driver converts input data to the PUT's arguments. After that, a RAC validates each test's inputs using the PUT's precondition. If RAC validates the input tests, then the PUT will be run, while RAC checks the PUT for any assertion violations. RAC will send a signal to the CGF tool if it reveals a semantic bug in the PUT using the generated test, and such input test data are saved. Further, the CGF actively monitors the PUT, seeking to generate tests that cover new, previously unexplored branches within the PUT.

4. Preliminary Study

We performed a preliminary study of our approach, comparing JMLKelinci, which is a CGF tool that uses JML's RAC to check input validity, using our approach. This study found that, our approach could detect bugs in 78 out of 84 buggy programs.

JMLKelinci (A. Nilizadeh, Leavens, & Păsăreanu 2021) uses a RAC tool (OpenJML (Cok 2010, 2011, 2021)) with a CGF (Kelinci (Kersten et al. 2017)) to provide valid inputs for Java programs. JMLKelinci obtains the benefits of using preconditions written in JML (Leavens et al. 2006, 2008, 2022) to cover branches with only valid inputs to provide the opportunity of using postconditions for revealing potential semantic bugs. JMLKelinci only uses the first step of this paper's idea by using formal preconditions for generating valid inputs; the JMLKelinci tool has not yet been adapted to investigate checking postconditions or to send feedback to the fuzzer regarding the presence of semantic bugs.

[5] The measure was called a "cost" in the paper about Kelinci-WCA (Noller et al. 2018; S. Nilizadeh et al. 2019), where the goal was to find the worst-case execution time of code in order to discover side-channels (Le et al. 2021). Using the measure in the opposite sense allows it to be used as a reward that is maximized.

[6] It may not be necessary for RAC to give a reward when input is valid; if the RAC uses the same code to reject an invalid input each time, then a CGF will try to avoid invalid inputs while attempting to cover more code.

4.1. Experimental Details

The experiment used three versions of 28 programs from the Buggy Java+JML dataset (A. Nilizadeh 2021a; A. Nilizadeh, Leavens, Le, et al. 2021; A. Nilizadeh et al. 2024), for a total of 84 programs.[7] Each buggy program in this dataset is a version of a correct program that has been proven correct using JML's tools. For each of the 28 correct programs, JMLKelinci was able to cover all branches with valid inputs.

In our experimental procedure, we used JMLKelinci, which effectively covers the branches of the buggy programs by generating valid inputs. It is worth mentioning that JMLKelinci saves the first input that either triggers a crash or explores a new branch within the code. Following the generation of these crucial inputs, we proceeded to apply OpenJML's RAC manually. This step was integral in our process as it allowed us to uncover and understand the presence of semantic bugs within the set of valid inputs that were saved previously.

4.2. Results

Table 1 shows the results of running both JMLKelinci alone and using JMLKelinci and RAC to check postconditions on these 84 buggy programs. In Table 1, the "JK" stands for JMLKelinci and "JKP" stands for JMLKelinci plus postcondition checking on the saved valid inputs. Also, "miss" means that the bug was not revealed with the generated inputs, and "detect" means that the bug was detected with valid inputs.

Our preliminary study indicates that using postcondition checking with RAC has a significant advantage in bug detection, revealing 78 out of 84 bugs when applied to the saved, valid inputs. On the contrary, the standalone use of the fuzzer JMLKelinci revealed only 26 bugs, which all caused program crashes. Therefore, incorporating postconditions and RAC in the process offers a superior approach to discovering semantic bugs, compared to relying only on a fuzzer.

Figure 3 shows a buggy variant of the `PrimeCheck` class, discussed in §2.3. This figure corresponds to the dataset's second buggy version. A bug is introduced at line 12, where the code is changed to `int mid = a*2;`. JMLKelinci, only validating preconditions, enables valid inputs to execute the program and stores those that cover a new branch or trigger a crash in the PUT. Even though the program displayed in Fig 3 has a bug, JMLKelinci fails to identify it as a bug because no input can lead the buggy program to crash during execution. Our preliminary study used RAC, and when it received a valid input (a six-digit number) generated by JMLKelinci, the tool saved this input, since it covered a new branch. RAC then confirmed the bug's presence within the program by indicating a postcondition violation.

Moreover, out of the 84 buggy programs, six cases failed to reveal the bug despite having a valid input covering the faulty branch. A more comprehensive approach involving automatic postcondition checking via RAC for all generated valid inputs - not just the first valid input that traverses a branch and is subsequently saved (as done in this study) - could potentially reveal these undetected bugs.

[7] We used buggy versions 1, 2, and 3 of each correct program. All programs in this dataset all have JML specifications.

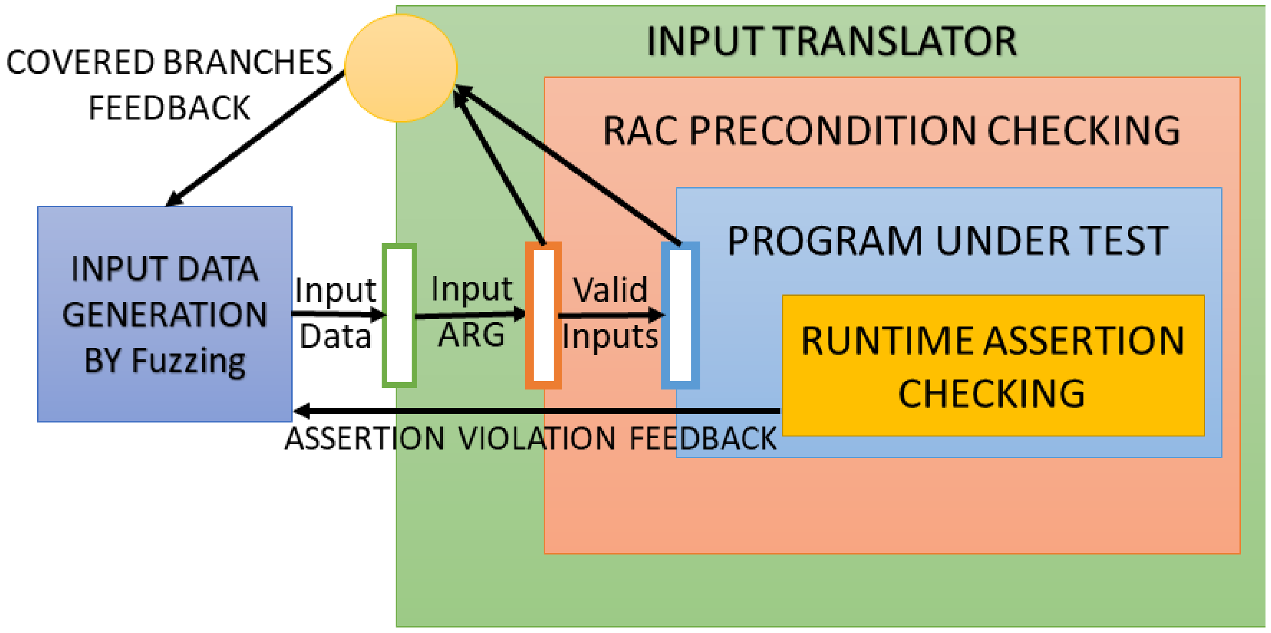
COVERED BRANCHES FEEDBACK
INPUT TRANSLATOR
RAC PRECONDITION CHECKING
PROGRAM UNDER TEST
INPUT DATA GENERATION BY Fuzzing
Input Data
Input ARG
Valid Inputs
RUNTIME ASSERTION CHECKING
ASSERTION VIOLATION FEEDBACK

Figure 2 Architecture of CGF with user-defined rewards using RAC.

```
class PrimeCheck {
   /*@    requires d != 0;
     @    ensures \result == n%d; @*/
    public /*@ pure @*/ static int
        div(int n, int d) { return n%d; }

   /*@ requires 1 < a;
     @ ensures \result <==> (\forall int k;
       1 < k && k <= a/2; div(a, k) != 0); @*/
   public boolean isPrime(int a) {
    int i = 2;
    int mid = a*2;  // bug on this line!
    while (i <= mid) {
      if (div(a,i) == 0)
        return false;
      i++;
    }
    return true;
   }
}
```

Figure 3 A buggy version of the PrimCheck program with JML annotations.

Figure 4 shows the buggy part of one of the six problematic programs from our set, specifically, the first faulty iteration of `CombinationPermutation` from the Java+JML dataset. This program aims to compute the number of combinations of two input parameters, `n` and `r`. In the correct version, line seven should read as follows.

```
combin = fac.factorial(n) / (fac.
    factorial(r) * fac.factorial(n-r));
```

Notably, the Factorial program, also annotated with JML, is called within this computation. Execution with JMLKelinci, an approach developed to identify crashes and achieve complete branch coverage, reveals no crash, since this bug does not generate crashes. Afterward, we used RAC using the generated inputs and postconditions to cover contract violations. However, the generated inputs that resulted in branch coverage were `n == 0` and `r == 0`, which do not lead to contract violations in this buggy program. Recall that JMLKelinci saves inputs when they cover a previously unexplored branch, as these values did. Therefore, even if JMLKelinci runs this program's branch with other inputs, those inputs are not saved because they do not cover a new branch in the program. This scenario stresses the potential value of a tool capable of automatically checking postconditions for all valid inputs, which could effectively reveal bugs like this one.

The results in Table 1 indicate that when JMLKelinci detected a bug, JMLKelinci with postconditions could also detect it. Also, when JMLKelinci with postconditions could not reveal a bug, JMLKelinci alone could not reveal the bug either. The total number of detected buggy programs using JMLKelinci alone and JMLKelinci with postconditions (by using the same driver)

Table 1 JMLKelinci (JK) and JMLKelinci + Postcondition (JKP) results on programs from the Buggy Java+JML Dataset.

Name	Bug1 JK	Bug1 JKP	Bug2 JK	Bug2 JKP	Bug3 JK	Bug3 JKP
Absolute	miss	detect	miss	detect	miss	detect
AddLoop	miss	detect	miss	detect	miss	detect
Alphabet	miss	detect	miss	detect	miss	detect
BankAccount	miss	detect	miss	detect	miss	detect
BinarySearch	miss	detect	detect	detect	detect	detect
BubbleSort	detect	detect	miss	detect	miss	detect
Calculator	miss	detect	miss	miss	detect	detect
CombinaPerm	miss	miss	detect	detect	miss	detect
CopyArray	miss	detect	detect	detect	detect	detect
Factorial	miss	detect	miss	detect	miss	detect
Fibonacci	detect	detect	detect	detect	detect	detect
FindFirstSorted	miss	miss	miss	detect	detect	detect
FindFirstZero	miss	detect	detect	detect	detect	detect
FindInArray	miss	detect	miss	miss	miss	miss
FindInSorted	miss	detect	miss	detect	detect	detect
GCD	miss	detect	miss	detect	miss	detect
Inverse	miss	detect	detect	detect	miss	detect
LCM	miss	detect	miss	detect	miss	detect
LeapYear	miss	detect	miss	detect	miss	miss
LinearSearch	detect	detect	miss	detect	miss	detect
OddEven	miss	detect	miss	detect	miss	detect
Perimeter	miss	detect	miss	detect	miss	detect
PrimeCheck	miss	detect	miss	detect	miss	detect
Smallest	miss	detect	detect	detect	detect	detect
StrPalindrome	miss	detect	detect	detect	detect	detect
StudentEnrollment	detect	detect	detect	detect	detect	detect
Time	detect	detect	miss	detect	miss	detect
TransposeMatrix	miss	detect	detect	detect	detect	detect

```
//@ requires 0<=n && n<=20 && 0<=r && r<=n;
//@ old Factorial fac_spec = new Factorial();
//@ ensures \result == fac_spec.spec_factorial
    (n) / (fac_spec.spec_factorial(r) *
    fac_spec.spec_factorial(n-r));
private /* pure @*/ long comb(int n, int r) {
    Factorial fac = new Factorial();
    long combin;
    combin = fac.factorial(n) / (fac.factorial
    (r) * fac.factorial(n+r));
    return combin;
}
```

Figure 4 Buggy Java program with JML annotations that RAC did not detect the bug.

for each buggy program is shown in Table 2. These results show that JMLKelinci alone could reveal about 31% of bugs in these programs, while using RAC for checking postcondition using saved generated inputs could reveal about 93% of bugs.

Table 2 Summary of bug detection results for JMLKelinci (JK) and JMLKelinci + Postconditions (JKP) on programs from the Buggy Java+JML Dataset.

Bug	Detected by JK		Detected by JKP	
	Num.	Percent	Num.	Percent
Version1	5	18%	26	93%
Version2	10	36%	26	93%
Version3	11	39%	26	93%
Total	26	31%	78	93%

4.3. Threats to Validity

An important threat to validity is that the programs in our initial experiments were fairly small and not drawn from real-world examples; thus they may not be a good guide to techniques for finding bugs in more complex, real-world programs. However, each of the 84 programs, although small, exhibits substantial complexity. This threat highlights an area for future work. Specifically, to enhance the robustness of our experiments, it is important to extend the evaluation to larger, more real-world programs in the future.

5. Related Work

Many approaches have been used for generating tests to reveal bugs. However, there is not much research on combining a fuzzer and a RAC tool in software testing. The work of Peters and Parnas (Peters & Parnas 1994) explained how to use a RAC in testing, but their work, and the work of Cheon and Leavens (Cheon & Leavens 2002, 2004), Zimmerman and Nagmoti (Zimmerman & Nagmoti 2010), and Xu and Yang (Xu & Yang 2003), did not use a fuzzer and so requires the user to create input test data. Some of the above works can automatically provide a small number of tests; however, a user must manually provide better input data to reveal semantic bugs. Thus, the effectiveness of RAC for detecting bugs in the above works depends on the quality of the input data provided. Without appropriate input data, a RAC tool might not reveal some bugs. In contrast, in our approach a fuzzer automatically supplies test data, and when used with a CGF our approach can also provide better coverage.

Several tools use model checking and symbolic execution to generate input tests and discover semantic bugs (Visser et al. 2004; Chipounov et al. 2009; Cadar et al. 2011; Nayrolles et al. 2015, 2017; F. Nilizadeh et al. 2023). However, the execution time of these techniques depends on the size of the PUT; thus, they have limited scalability in comparison to fuzzing techniques (Ognawala et al. 2018).

Korat (Boyapati et al. 2002; Milicevic et al. 2007) uses formal specifications to generate test data and decide whether tests pass. However, Korat does not try to improve code coverage.

EvoSuite (Fraser & Arcuri 2011, 2013), Randoop (Pacheco et al. 2007), and TSTL (Groce et al. 2015; Holmes et al. 2018) are other tools that can provide unit tests for a program using its code and assertions. However, they do not guarantee that the inputs to the PUT are valid.

Property-based testing (PBT) (Fink & Bishop 1997; Paraskevopoulou et al. 2015) is a different technique that seeks to identify property violations by generating random test cases. A special implementation of PBT is QuickCheck in Haskell (Claessen & Hughes 2000). The PBT method uses randomness in test case generation without giving feedback to the random test generator. Thus, it is similar to using black-box fuzzing with RAC to reveal semantic bugs. However, it does not try to optimize code coverage in the PUT, as would be the case in our approach with a CGF tool.

One of the closest works to our study is JMLKelinci (A. Nilizadeh, Leavens, & Păsăreanu 2021), which uses RAC to help a CGF tool generate valid inputs to cover branches for Java programs. However, JMLKelinci did not use postconditions to reveal semantic bugs or send feedback to help the fuzzer. This paper shows how the idea of tools like JMLKelinci can be extended to reveal semantic bugs.

The work most analogous to our concept is JMLKelinci+ (A. Nilizadeh et al. 2023). In that work, the first idea of our paper has been developed for Java programs. The approach employed a CGF tool for input generation. Subsequently, a RAC tool was used to filter out invalid inputs based on preconditions, and after that, it checks postconditions and internal assertions to reveal semantic bugs. Nonetheless, the other ideas of integrating RAC with hybrid fuzzers and CGF with user-defined costs remain unexplored.

Our approach of using a RAC to check for valid inputs is inspired by what some programs do when sanitizing inputs (e.g., to prevent code injection attacks) (Wang et al. 2013; Serebryany 2016; Österlund et al. 2020). As in that approach, assertion checking is more automatic than standard input/output testing, and it has better reusability and maintainability. The main difference between our approach and sanitizer approaches is that the PUT will only execute with valid inputs in our approach; thus, when a postcondition violation is detected, the tool has truly found a semantic bug. Furthermore, an assertion violation in sanitizer leads to a program crash. However, in our approach, the input that causes an assertion violation is saved as an input test.

Another related work is the Universal Verification Methodology (UVM) (Height 2010; Mehta 2018), an IEEE standard for hardware testing (design verification). UVM is a mature methodology that is supported by industry tools, and semiconductor companies use UVM as one of the main steps for detecting bugs in hardware designs. UVM generates random input tests based on the preconditions (known as Constrained Random Verification), and then it monitors the behavior of the design using checkers (which are similar to the code that RAC generates to check inner and postconditions assertions). The core concept presented in this idea paper is reminiscent of UVM. However, it is important to note that UVM is primarily used for detecting anomalies and bugs in hardware designs that are articulated through hardware description languages, rather than revealing bugs within software.

6. Patch Correctness Assessment for APR

Automated program repair (APR) has promising results for repairing buggy programs. In the absence of formal verification, most software APR tools use test suites to detect bugs and validate candidate patches' correctness. Due to the incompleteness of test suites, repairs generated by APR tools can be incorrect based on the expected behavior of the correct program. This issue is referred to as "test overfitting" in APR (Smith et al. 2015; Le et al. 2018; A. Nilizadeh, Leavens, Le, et al. 2021; A. Nilizadeh 2022). Thus, it is necessary to evaluate the correctness of the generated patches.

One recent approach in APR is generating many repaired programs, all of which pass the program's test suite. This gives developers more options to compare patches and select the best one. However, sometimes APR tools generate thousands of candidate patches (Yuan & Banzhaf 2018), which makes it tedious to pick the best patches manually. A potential solution to this problem could be the application of fuzzing techniques aimed at automatically discarding overfitted patches. On the other hand, this problem may justify the cost of adding lightweight pre- and postconditions to programs, as then one can use our approach to eliminate overfitted patches automatically. Also, RAC violations show a weakness in the input test suite that the absence of that test leads to generating test overfitted patches by APR. Adding those generated input tests to the input test suite can cover those weaknesses. Then, by running the APR, they can generate more reliable patches (A. Nilizadeh, Calvo, et al. 2021; Huang & Meyer 2022; A. Nilizadeh 2021b).

Additionally, the concept we discussed regarding APR patch correctness could also be applied to automated reasoning repair (A. Nilizadeh, Leavens, & Cok 2022; Ringer et al. 2018); this is particularly feasible given that the specification is readily available.

7. Limitations of Our Approach

The main limitation of combining RAC and fuzzing is the need for lightweight formal specifications. If there are no formal specifications, then lightweight specifications, at least for the entry method of the PUT, must be written. However, for programs that already have such formal specifications, such as critical systems and security protocol implementations, our approach could be easily implemented.

Another limitation of our approach is that the specifications may not be adequate; the pre- and postcondition specified for the PUT's entry method might be inadequate, or they might not be able to be evaluated with RAC. If the precondition of the PUT's entry method is too strong, then it might prevent some inputs that should be allowed from being used in tests. Conversely, if the postcondition is too weak or cannot be fully evaluated using RAC, then it might miss some bugs that it should catch. In both cases, a tool using our approach might not be able to reveal semantic bugs in the PUT.

In short our approach depends on having accurate pre- and postconditions for the PUT's entry method that can be evaluated by RAC.

8. Conclusions

We have proposed the idea of combining modern fuzzing tools with lightweight specifications checked by runtime assertion checking (RAC) tools; this combination can help to reveal potential semantic bugs automatically. Fuzzing alone would only generate inputs (perhaps invalid) that lead to a program crash, but the combination of a fuzzer and RAC can generate and check for valid inputs (using preconditions) and then check for semantic bugs (using postconditions).

We have also discussed how a modern coverage-guided fuzzer (CGF) can best be combined with RAC. RAC can check preconditions to ensure that the PUT is only run with valid inputs, which saves testing time. Checking postconditions with RAC can help guide a CGF, helping it generate better inputs for testing. Our preliminary study on 84 buggy programs shows that combining a fuzzer and and RAC can lead to effective tools for revealing semantic bugs; this study showed that this combination discovered bugs in 78 of 84 buggy programs (93%). In contrast, fuzzing with valid inputs (obtained by checking inputs against the PUT's preconditions using RAC) could only reveal 26 bugs in 84 programs (31%).

Acknowledgments

We extend our sincere appreciation to Corina Păsăreanu, Yannic Noller, and Shirin Nilizadeh for their valuable insights and discussions.

References

Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., ... others (2014). The key platform for verification and analysis of java programs. In *Verified software:*

Theories, tools and experiments: 6th international conference, vstte 2014, vienna, austria, july 17-18, 2014, revised selected papers 6 (pp. 55–71).

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., & Ulbrich, M. (2016). Deductive software verification-the KeY book. *Lecture notes in computer science*, *10001*.

Alur, R., Singh, R., Fisman, D., & Solar-Lezama, A. (2018). Search-based program synthesis. *Communications of the ACM*, *61*(12), 84–93.

Apt, K. R., & Olderog, E.-R. (2019). Fifty years of Hoare's logic. *Formal Aspects of Computing*, *31*(6), 751–807.

Barner, S., Glazberg, Z., & Rabinovitz, I. (2005). Wolf–bug hunter for concurrent software using formal methods. In *International Conference on Computer Aided Verification* (pp. 153–157).

Boehme, M., Cadar, C., & Roychoudhury, A. (2021). Fuzzing: Challenges and reflections. *IEEE Softw.*, *38*(3), 79–86.

Boyapati, C., Khurshid, S., & Marinov, D. (2002). Korat: Automated testing based on Java predicates. *ACM SIGSOFT Software Engineering Notes*, *27*(4), 123–133.

Braione, P., Denaro, G., Mattavelli, A., & Pezzè, M. (2017). Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 90–101).

Cadar, C., Dunbar, D., Engler, D. R., et al. (2008). KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (Vol. 8, pp. 209–224).

Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C. S., Sen, K., Tillmann, N., & Visser, W. (2011). Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)* (pp. 1066–1071).

Cha, S. K., Avgerinos, T., Rebert, A., & Brumley, D. (2012). Unleashing Mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (p. 380–394). USA: IEEE Computer Society.

Chalin, P., Kiniry, J. R., Leavens, G. T., & Poll, E. (2006). Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal methods for components and objects (fmco) 2005, revised lectures* (Vol. 4111, p. 342-363). Berlin: Springer-Verlag. Retrieved from https://tinyurl.com/3z2vk55n

Chang, O., Arya, A., Serebryany, K., & Armour, J. (2017, May). *Oss-fuzz: Five months later, and rewarding projects.* https://testing.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html. (Accessed: 2023-03-15)

Cheon, Y., & Leavens, G. T. (2002, June). A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson (Ed.), *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, proceedings* (Vol. 2374, p. 231-255). Berlin: Springer-Verlag. Retrieved from https://tinyurl.com/4tk2nzzd

Cheon, Y., & Leavens, G. T. (2004). The JML and JUnit way of unit testing and its implementation. *Technical Report TR# 04-02a, Department of Computer Science*.

Chipounov, V., Georgescu, V., Zamfir, C., & Candea, G. (2009). Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*.

Claessen, K., & Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth acm sigplan international conference on functional programming* (pp. 268–279).

Clarke, L. A. (1976). A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*(3), 215–222.

Clarke, L. A., & Rosenblum, D. S. (2006). A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, *31*(3), 25–37.

Cok, D. R. (2010). Improved usability and performance of SMT solvers for debugging specifications. *International Journal on Software Tools for Technology Transfer*, *12*(6), 467–481.

Cok, D. R. (2011). OpenJML: JML for Java 7 by extending OpenJDK. In *NASA Formal Methods Symposium* (pp. 472–479).

Cok, D. R. (2021). JML and OpenJML for Java 16. In *Proceedings of the 23rd acm international workshop on formal techniques for java-like programs* (p. 65–67). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3464971.3468417

Filliâtre, J.-C., & Pascutto, C. (2021). Ortac: Runtime assertion checking for OCaml (tool paper). In *International Conference on Runtime Verification* (pp. 244–253).

Fink, G., & Bishop, M. (1997). Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, *22*(4), 74–80.

Fraser, G., & Arcuri, A. (2011). EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (pp. 416–419).

Fraser, G., & Arcuri, A. (2013). EvoSuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* (pp. 362–369).

Gazzola, L., Micucci, D., & Mariani, L. (2018). Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 1219–1219).

Godefroid, P., Levin, M. Y., & Molnar, D. (2012). SAGE: whitebox fuzzing for security testing. *Queue*, *10*(1), 20–27.

Goues, C. L., Pradel, M., & Roychoudhury, A. (2019). Automated program repair. *Communications of the ACM*, *62*(12), 56–65.

Groce, A., Pinto, J., Azimi, P., & Mittal, P. (2015). TSTL: a language and tool for testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (pp. 414–417).

Height, H. (2010). *A practical guide to adopting the universal verification methodology (UVM)*. Lulu. com.

Hoare, C. A. R. (1969, October). An axiomatic basis for computer programming. *Communications of the ACM*, *12*(10), 576–580,583. Retrieved from http://doi.acm.org/

10.1145/363235.363259

Hoare, C. A. R. (1972). Proof of correctness of data representations. *Acta Informatica*, *1*(4), 271-281. Retrieved from http://dx.doi.org/10.1007/BF00289507

Holmes, J., Groce, A., Pinto, J., Mittal, P., Azimi, P., Kellar, K., & O'Brien, J. (2018). TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, *20*(1), 57–78.

Huang, L., & Meyer, B. (2022). A failed proof can yield a useful test. *arXiv preprint arXiv:2208.09873*.

Jiang, L., Yuan, H., Wu, M., Zhang, L., & Zhang, Y. (2023). Evaluating and improving hybrid fuzzing. In *2023 IEEE/ACM 45th international conference on software engineering (ICSE)* (pp. 410–422).

Kersten, R., Luckow, K., & Păsăreanu, C. S. (2017). Poster: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2511–2513).

Kosmatov, N., Maurica, F., & Signoles, J. (2020). Efficient runtime assertion checking for properties over mathematical numbers. In *International Conference on Runtime Verification* (pp. 310–322).

Le, X.-B. D., Pasareanu, C., Padhye, R., Lo, D., Visser, W., & Sen, K. (2021). Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes*, *44*(4), 14–14.

Le, X.-B. D., Thung, F., Lo, D., & Goues, C. L. (2018). Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 163–163).

Leavens, G. T., Baker, A. L., & Ruby, C. (2006, March). Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, *31*(3), 1-38. Retrieved from http://doi.acm.org/10.1145/1127878.1127884

Leavens, G. T., Cok, D. R., & Nilizadeh, A. (2022). Further lessons from the JML project. In *The Logic of Software. A Tasting Menu of Formal Methods* (pp. 313–349). Springer.

Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D. R., ... Zimmerman, D. M. (2008, May). *JML Reference Manual.* (Available from http://www.jmlspecs.org)

Leucker, M., & Schallhart, C. (2009). A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, *78*(5), 293–303.

Li, J., Zhao, B., & Zhang, C. (2018). Fuzzing: a survey. *Cybersecurity*, *1*(1), 6.

Liang, H., Pei, X., Jia, X., Shen, W., & Zhang, J. (2018). Fuzzing: State of the art. *IEEE Transactions on Reliability*, *67*(3), 1199–1218.

Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., & Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*.

Mehta, A. B. (2018). UVM (Universal Verification Methodology). In *ASIC/SoC Functional Design Verification* (pp. 17–64). Springer.

Meyer, B. (1997). *Object-oriented software construction* (second ed.). New York, NY: Prentice Hall.

Microsoft Corp. (2015, January). *Microsoft security risk detection.* https://www.microsoft.com/en-us/security-risk-detection/. (Accessed: 2023-03-15)

Milicevic, A., Misailovic, S., Marinov, D., & Khurshid, S. (2007). Korat: A tool for generating structurally complex test inputs. In *29th International Conference on Software Engineering (ICSE'07)* (pp. 771–774).

Nayrolles, M., Hamou-Lhadj, A., Tahar, S., & Larsson, A. (2015). JCHARMING: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 101–110).

Nayrolles, M., Hamou-Lhadj, A., Tahar, S., & Larsson, A. (2017). A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process*, *29*(3), e1789.

Nilizadeh, A. (2021a). *BuggyJavaJML*. https://github.com/Amirfarhad-Nilizadeh/BuggyJavaJML. (Accessed: 2023-03-15)

Nilizadeh, A. (2021b). *Test overfitting: Challenges, approaches, and measurements* (Tech. Rep.). University of Central Florida, Computer Science.

Nilizadeh, A. (2022). Automated program repair and test overfitting: Measurements and approaches using formal methods. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 480–482).

Nilizadeh, A., Calvo, M., Leavens, G. T., & Cok, D. R. (2022). Generating counterexamples in the form of unit tests from Hoare-style verification attempts. In *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering* (pp. 124–128).

Nilizadeh, A., Calvo, M., Leavens, G. T., & Le, X.-B. D. (2021). More reliable test suites for dynamic APR by using counterexamples. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)* (p. 208-219). IEEE.

Nilizadeh, A., & Leavens, G. T. (2022). Be realistic: Automated program repair is a combination of undecidable problems. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)* (pp. 31–32).

Nilizadeh, A., Leavens, G. T., & Cok, D. R. (2022). Automated reasoning repair. In *Proceedings of the 24th acm international workshop on formal techniques for java-like programs* (pp. 11–14).

Nilizadeh, A., Leavens, G. T., Le, X.-B. D., Păsăreanu, C. S., & Cok, D. R. (2021). Exploring true test overfitting in dynamic automated program repair using formal methods. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)* (pp. 229–240).

Nilizadeh, A., Leavens, G. T., & Păsăreanu, C. S. (2021). Using a guided fuzzer and preconditions to achieve branch coverage with valid inputs. In *International Conference on Tests and Proofs* (pp. 72–84).

Nilizadeh, A., Leavens, G. T., Păsăreanu, C. S., Le, X.-B. D.,

& Cok, D. R. (2024). Does going beyond branch coverage make program repair tools more reliable? In *2024 17th IEEE Conference on Software Testing, Verification and Validation (ICST) (In Press).* IEEE.

Nilizadeh, A., Leavens, G. T., Păsăreanu, C. S., & Noller, Y. (2023). JMLKelinci+: Detecting semantic bugs and covering branches with valid inputs using coverage-guided fuzzing and runtime assertion checking. *Formal Aspects of Computing*.

Nilizadeh, F., Dashtbani, H., & Mouzarani, M. (2023). Parameterized search heuristic prediction for concolic execution. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC) (In Press).* IEEE.

Nilizadeh, S., Noller, Y., & Pasareanu, C. S. (2019). DifFuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 176–187).

Noller, Y., Kersten, R., & Păsăreanu, C. S. (2018). Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 322–332).

Noller, Y., Păsăreanu, C. S., Böhme, M., Sun, Y., Nguyen, H. L., & Grunske, L. (2020). Hydiff: Hybrid differential software analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (pp. 1273–1285).

Odena, A., Olsson, C., Andersen, D., & Goodfellow, I. (2019). Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning* (pp. 4901–4911).

Ognawala, S., Hutzelmann, T., Psallida, E., & Pretschner, A. (2018). Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (pp. 1475–1482).

Österlund, S., Razavi, K., Bos, H., & Giuffrida, C. (2020). Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)* (pp. 2289–2306).

Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. (2007). Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)* (pp. 75–84).

Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., & Pierce, B. C. (2015). Foundational property-based testing. In *International conference on interactive theorem proving* (pp. 325–343).

Peleg, H., & Polikarpova, N. (2020). Perfect is the enemy of good: Best-effort program synthesis. *Leibniz International Proceedings in Informatics*, *166*.

Peters, D., & Parnas, D. L. (1994, August). Generating a test oracle from program documentation. In *Proceedings of ISSTA 94, seattle, washington, august, 1994* (pp. 58–65). ACM Press.

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). VUzzer: Application-aware evolutionary fuzzing. In *NDSS* (Vol. 17, pp. 1–14).

Ringer, T., Yazdani, N., Leo, J., & Grossman, D. (2018). Adapting proof automation to adapt proofs. In *Proceedings of the 7th acm sigplan international conference on certified programs and proofs* (pp. 115–129).

Serebryany, K. (2016). Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)* (pp. 157–157).

Smith, E. K., Barr, E. T., Le Goues, C., & Brun, Y. (2015). Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 532–543).

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., … Vigna, G. (2016). Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS* (Vol. 16, pp. 1–16).

Taleb, R., Khoury, R., & Hallé, S. (2021). Runtime verification under access restrictions. In *Proceedings of the 9th International Conference on Formal Methods in Software Engineering.*

Visser, W., Păsăreanu, C. S., & Khurshid, S. (2004). Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (pp. 97–107).

Wang, X., Zeldovich, N., Kaashoek, M. F., & Solar-Lezama, A. (2013). Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (pp. 260–275).

Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, *42*(8), 707–740.

Xu, G., & Yang, Z. (2003). JMLAutoTest: A novel automated testing framework based on JML and JUnit. In *International workshop on Formal Approaches to Software Testing* (pp. 70–85).

Yuan, Y., & Banzhaf, W. (2018). Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on software engineering*, *46*(10), 1040–1067.

Yue, T., Tang, Y., Yu, B., Wang, P., & Wang, E. (2019). LearnAFL: Greybox fuzzing with knowledge enhancement. *IEEE Access*, *7*, 117029–117043.

Zalewski, M. (2014). Technical" whitepaper" for afl-fuzz. *URl: http://lcamtuf. coredump. cx/afl/technical_details.txt*.

Zimmerman, D. M., & Nagmoti, R. (2010). Jmlunit: The next generation. In *International Conference on Formal Verification of Object-Oriented Software* (pp. 183–197).

About the authors

Amirfarhad Nilizadeh has been working at the Advanced Micro Devices (AMD) formal verification team as a member of technical staff since January 2022. He earned his Ph.D. in Computer Science from the University of Central Florida (UCF) in December 2021, where he accumulated his research and teaching skills over a period of five years. His primary areas of interest encompass formal verification for both hardware designs and software programs, testing, automated program repair, and cybersecurity. In pursuit of applied experience, he completed an internship at AMD in the fall of 2021 and participated in the

Google Summer of Code later that summer. Additionally, in 2018, he accomplished a research position at the formal verification group of CyLab Security & Privacy Institute at Carnegie Mellon University. Before starting his Ph.D., he worked as a university lecturer at Azad University from 2014 to 2016, complementing his research with valuable teaching experience. You can contact the author at amirfarhad.nilizadeh@gmail.com.

Gary T. Leavens is a professor and former chair of the department of Computer Science at the University of Central Florida (UCF). Before joining UCF in fall 2007, he was a professor of Computer Science at Iowa State University in Ames, Iowa, where he started in January 1989, after receiving his Ph.D. from MIT. Before his graduate studies at MIT, he worked at Bell Telephone Laboratories in Denver Colorado as a member of technical staff. You can contact the author at Leavens@ucf.edu or visit `https://www.cs.ucf.edu/~leavens/`.

David R. Cok is a researcher in deductive reasoning, having contributed to the development and application of JML, OpenJML, Dafny, ACSL, Frama-C/C++, and SPARK for Ada. He previously was a researcher on digital imaging and automated reasoning at Kodak Research Laboratories and on program analysis at GrammaTech. He has a Ph.D. in Physics from Harvard University. You can contact the author at david.r.cok@gmail.com.