Finding Specification Blind Spots via Fuzz Testing

Ru Ji
University of Waterloo

Meng Xu
University of Waterloo

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179438

***Abstract*—A formally verified program is only as correct as its specifications (SPEC). But how to assure that the SPEC is complete and free of loopholes? This paper presents FAST, short for Fuzzing-Assisted Specification Testing, as a potential answer. The key insight is to exploit and synergize the "redundancy" and "diversity" in formally verified programs for cross-checking. Specifically, within the same codebase, SPEC, implementation (CODE), and test suites are all derived from the same set of business requirements. Therefore, if some intention is captured in CODE and test case but not in SPEC, this is a strong indication that there is a blind spot in SPEC.**

FAST examines the SPEC for incompleteness issues in an automated way: it first locates SPEC gaps via mutation testing, i.e., by checking whether a CODE variant conforms to the original SPEC. If so, FAST further leverages the test suites to infer whether the gap is introduced by intention or by mistake. Depending on the codebase size, FAST may choose to generate CODE variants in either an enumerative or evolutionary way. FAST is applied to two open-source codebases that feature formal verification and helps to confirm 13 and 21 blind spots in their SPEC respectively. This highlights the prevalence of SPEC incompleteness in real-world applications.

1. Introduction

Formal verification delivers high-assurance to computing systems by mathematically checking the *correctness* of a program, i.e., the behaviors of a program are fully confined by a desired set of properties—the specifications—described with a formal modeling language. Formal verification has seen its adoption in hardware and software systems for decades. Typical application scenarios include cases where errors may lead to significant losses and irreversible consequences (e.g., in the field of aviation [13, 49, 66, 82]) or in cases when the traditional rolling program upgrade model is not feasible such as smart contracts on blockchains [1, 67].

Abstractly, applying formal verification to a computing system can be decomposed into two (somewhat) orthogonal processes: ① developing a complete set of specifications for the target system and ② proving or disproving that the actual implementation is in conformance with the specifications. In this paper, we use SPEC as an abbreviation of the specifications modeled in predicate calculus with symbolic semantics and CODE to represent the actual implementation in programming languages with concrete and executable semantics. The formal verification process can then be decomposed into ① devising the SPEC and ② checking that SPEC $\sqsupseteq$ CODE (i.e., CODE conforms to SPEC).

Recent years have witnessed great progress on addressing problem ② as evident by the consistent stream of improvements on automated theorem provers [15, 33, 64, 74, 93, 94], while far less attention has been paid to problem ①. This can be a dangerous disparity—even a program is thoroughly verified with a perfect verification toolchain, this program is only as correct as its SPEC. Errors in the SPEC can be as bad, if not worse, as errors in the CODE. One of the concerning scenarios is unintended gaps in the SPEC. The gaps will create verification blind spots in which the program behaviors are unconstrained. In the worst case, there is nothing to prevent a malicious developer from hiding backdoors and trojans behind these blind spots [29, 30], and such malicious code can survive regardless of how rigorous the verification is—**a single blind spot in the SPEC can easily undermine months if not years of verification efforts**.

The consequences of an incomplete set of SPEC is exacerbated by the high costs of adopting formal methods. As of now, formal verification is still an expensive technique due to the extra effort of writing SPEC. In both case studies covered in the paper (§4.1 and §5.1), the SPEC is not developed by the team who originally write the CODE. Instead, they are developed by a dedicated team of experts with years of training and practice (including a PhD degree) in formal methods. However, despite the high costs, the industry is willing to pursue this route with an expectation that formally verified programs have higher assurance. While it is true that formally verified programs generally have higher assurance, it is important to boost a general awareness that the formally verified "stamp" should not be blindly trusted without a good understanding of the completeness of the SPEC in the first place.

Fortunately, *gauging the completeness of* SPEC is not a new problem—it has received more attention in hardware verification than software verification, likely because formal methods have a longer history in hardware design. Most of the existing solutions in hardware verification to detect incompleteness in SPEC are based on mutation testing [28, 84], where a mutant is created by altering either the SPEC or CODE and check if the mutant can be "killed", i.e., the mutated CODE or SPEC can be proved to be non-conformant with the unmodified counterpart. As a result, any surviving mutant raises a signal where the SPEC might be incomplete.

The mutation testing technique sheds light on how we might find gaps in the SPEC of formally verified software sys-

DOI 10.1109/SP46215.2023.00173

tems. In particular, it is natural to research on ❶ whether mutation testing is readily applicable in the software verification context; and ❷ if not, what improvements should be applied on conventional mutation testing. With an enhanced mutation testing framework oriented towards software verification, we can finally pursue our meta-quest: ❸ is incompleteness issues in SPEC prevalent in mature codebases?

In this paper, we seek to answer all questions raised above with an integrated tool: FAST, short for Fuzzing-Assisted Specification Testing. In particular, we first confirm that adopting mutation testing in the software verification context can be effective in uncovering SPEC gaps in our case studies—a sizable basket of low-hanging fruits. However, in the face of complicated programs, conventional mutation testing with a random mutation strategy is less effective in finding "deeper" and "more interesting" gaps in the SPEC.

To set the context for this paper, consider a procedure in which we attempt to measure SPEC completeness by producing a stream of CODE mutants and checking whether these mutants can be "killed" by the original SPEC. There are at least two challenges in this procedure:

- **When a mutant passes the verification, how can we tell that the gap in the SPEC is by intention or by mistake?** Even though the mutant passes the verification, it is still possible that the CODE is meant to be written freely in the unspecified part, which means the mutant does not indicate an underlying mistake, and the SPEC is intentionally abstract in this part of programming. Therefore, we need a method that *automatically* categorizes whether a gap in the SPEC is intentional or mistaken.
- **How to produce a mutant that is more likely to pass the verification?** Brute-force enumeration of all possible mutations in the CODE might work for simple software/hardware systems (e.g., UART circuit [69]), but such a practice can be futile in complicated software programs with nearly infinite ways to mutate. We need a systematic approach to produce "meaningful" mutants that can pass verification in a reasonable amount of time.

To tell whether a gap in the SPEC is by intention or by mistake, the fundamental insight is to exploit and synergize the "redundancy" and "diversity" in formally verified programs. To be exact, SPEC, CODE, and test suites are all derived from the same set of requirements but programmed with different mentalities: for example, in different languages (or even programming paradigms), asynchronously, with different evolution paths, and ideally by different and independent teams. It is therefore less likely that the three derivations will bear the same mistake. This paves the way for finding errors in one component by cross-comparing it against the other two. In fact, this principle is already applied to check the correctness of CODE by both running it against test suites and proving it against the SPEC. In this paper, we show that the same procedures can be used to check incompleteness in SPEC as well (and deficiencies in test suites too, as by-products of our methods).

We further solve the mutant generation problem with an *evolution* strategy adopted from modern software fuzzers. The insight is to simulate the natural selection process by allowing the mutant with higher "fitness" score to have more chances of further mutations. In essence, each CODE mutant is evaluated for "fitness" when verified against the SPEC and only high-quality mutants survive and participate in future rounds of mutation. In this way, all fuzzing efforts are retained, and each generation of mutants gets closer to the evolution goal—passing the verification. The "fitness" metric can be as simple as the number of verification errors triggered when verifying the CODE mutant against the SPEC.

Like most fuzz-based tools, FAST cannot guarantee the absence of incompleteness issues in SPEC, but can be used to boost confidence that there are no *obvious* loopholes in SPEC. In other words, we see FAST as a cheap but effective fortification on the financial and time investment on writing SPEC and also the co-evolution of SPEC and CODE, such that the accumulated formal verification effort can not be easily undermined by unintended omissions by SPEC writers.

Summary - This paper makes the following contributions:

- **Concept.** We point out the "redundancy" and "diversity" of SPEC, CODE, and test suites in formally verified programs and exploit this redundancy to solve the problem of judging whether a gap in the SPEC is intentional or mistaken by inviting the test suites as a "referee".
- **Design.** We enhance the conventional mutation testing practices with an evolutionary feedback loop that guides the production of high quality mutants. Like evolution, each batch of new mutants are increasingly harder to be "killed" by the SPEC until a surviving mutant is found.
- **Impact.** We found 13 and 21 blind spots in the SPEC for DPN and S2N (two case studies in this paper, details will be provided later), respectively. Our findings are concerning and we hope this can be helpful in drawing attention to the quality of SPEC in formally verified codebases.

2. Background and Related Work

In this section, we give a brief introduction to formal verification and the SPEC incompleteness problem. We then introduce and differentiate two stochastic testing methods—mutation testing and fuzz testing—which are later combined in our work. In light of the proliferation of research works in mutation and fuzz testing, this section also serves as a best-effort survey of related works with elaborations on how FAST differ from them.

2.1. Formal verification

Formal methods have been widely used in hardware and software systems such as CPU design (e.g., Intel floating-point instructions [39]), cryptographic libraries (e.g., HACL* [98]), operating systems (e.g., seL4 [48]), compilers (e.g., CompCert [53]), networking protocols (e.g., Amazon S2N-TLS [80]), and more recently, smart contracts (e.g., the Diem Payment Network [2]), etc. In these applications, formal verification provides arguably the highest level of

assurance anyone can claim to the safety and correctness of these systems.

Despite the vast number of flavors in formal methods, in general, applying verification to a system involves two (somewhat) orthogonal steps: ① devising a set of *specifications* (SPEC), often developed in an abstract logic language, to describe the desired properties; and ② developing *verification tools* to reason about the relationship between the SPEC and the target implementation (i.e., the CODE) that follows concrete and even executable semantics [46].

While much attention has been paid to ②—automated verification tools—the effectiveness and practical value of formal verification, however, is largely determined by the quality of the SPEC that confines the behaviors of a system. An ideal set of SPEC needs to be *complete* enough to capture all intentions from stakeholders and yet *abstract* enough to allow flexibility in implementation choices. For example, if the requirement is sorting an array, the SPEC needs to be *complete* enough to capture the semantics of sorting and yet *abstract* enough to allow both the quick sort and merge sort implementation to pass.

However, developing a high-quality set of SPEC is hard. A set of SPEC that shadows the CODE is of little practical value and only bloats the codebase, leading to both frictions on CODE changes and higher maintenance costs. On the other hand, if the SPEC is too abstract, it might fail to capture some of the essential design requirements, leading to potential vulnerabilities undetected in the CODE.

Unfortunately, despite the importance of SPEC, there are limited research works that objectively measure its quality other than using hand-tuned heuristics-based checklists [14, 54, 66]. This is in sheer contrast with the general perception of test suites (e.g., unit tests or end-to-end tests). The de facto standard to measure the quality of a test suite is code coverage and this metric has reached consensus in the open-source community. While this paper does not target at proposing a quality measurement metric for the spec, our results highlight the importance of such a metric and shed light on what can be useful elements in this metric. i.e., SPEC coverage, or more specifically, how a piece of code contribute to the establishment of a property in the SPEC.

2.2. Automated function verification

While FAST can be applied to different flavors of formal verification (e.g., protocol verification [40], state-machine transitions [23, 63], etc.), in this paper, we focus on a specific type of verification: functional correctness verification with preconditions and postconditions, sometimes also known as "design-by-contract" [61].

In function verification, the SPEC target is typically the CODE that constitute a single function and developers provide pre- and post-conditions for the function body in the form of SPEC predicates, which typically include conditions over function parameters and/or environmental states that can be referred to by the CODE in the function. The SPEC may include constructs that do not have concrete executable semantics, such as universal and existential quantification

```
fn add1(v: [int]) -> [int] {
  for i in 0..len(v) {
    v[i] = v[i] + 1;
  }
  return v;
}
```

(a) CODE for `add1`

```
spec for fn add1 {
  ensures forall
    i in 0..len(result):
      result[i] = v[i] + 1;
}
```

(b) SPEC for `add1`

Figure 1: Demonstration of potential incompleteness in SPEC

over unbounded domains. Although specified against a single function, pre- and post-conditions are not limited to establishing the correctness of one function only. They contribute to the establishment of overall program correctness as preconditions are verified at caller side such that postconditions can be assumed after the call.

Recent years have seen a gradual adoption of many function verification frameworks and broadly categorized, they follows either *automated deductive verification* in which the manual effort is limited to writing the SPEC only and the proof obligation is fulfill automatically (e.g., SeaHorn [37], Kani [3] VeriFast [43]), or *interactive verification* in which both the SPEC and a majority of the proof needs to be developed manually (e.g., HOL [35], Isabelle [92], Coq [42]). It is worth highlighting that FAST requires a fully automated process on checking whether a CODE mutant conforms to SPEC. Therefore, FAST is only compatible with automated deductive function verification systems.

Figure 1 is an illustration of function verification. In this simple case, the developers' intention, as correctly implemented in the CODE, is to increment one for each element in the vector. The SPEC for this function, as described in the `ensures` postcondition, asserts that for each element in the return vector, it gets incremented by one compared with its counterpart in the input vector. The SPEC shows no preconditions, as this function can be called from any state. It is then the job of the verification tool to fuse the CODE and SPEC into a proof obligation that can be discharged to backend solvers (typically SMT solvers) to handle.

2.3. The completeness of specifications

While it is obvious that in Figure 1 the CODE conforms to the SPEC, the SPEC, however, has a serious omission and does not fully capture the developers' intention. Imagine if the `add1` function is implemented differently, as shown in Figure 2a, with an extra `pop()` after the original loop. The current SPEC, which only checks whether the value of every *remaining* element in the vector is increased by one, will still pass under the code mutant—an undesirable behavior! The complete spec is shown in Figure 2b with an extra `ensures` clause which further restricts the ability for the `add1` function to modify the input vector. This missing `ensures` represents an incompleteness issue of the original SPEC.

The example in Figure 1 and 2 highlights a lesser-known view about formal verification—writing SPEC is essentially another form of programming to capture the same requirement, just like writing CODE [68]. Therefore, if bugs are commonly found in CODE, especially in large codebases, how can we be assured that the SPEC is not "buggy"? The idea of finding discrepancies between SPEC and CODE is known as

```
fn add1(v: [int]) -> [int] {
  for i in 0..len(v) {
    v[i] = v[i] + 1;
  }
  // mutation to the code
  v.pop();
  return v;
}
```
(a) CODE mutant for add1

```
spec for fn add1 {
  ensures forall
    i in 0..len(result):
      result[i] = v[i] + 1;
  // missed post-condition
  ensures
    len(result) == len(v);
}
```
(b) Complete SPEC for add1

Figure 2: Finding the gap in SPEC via CODE mutant

gauging the completeness of SPEC in the literature [79, 85], and has received more attention in hardware (integrated circuit in particular) verification than software verification, likely due to the fact that formal methods have a longer history in hardware design. A recent work uses an inductive way to show the gap between the mathematical model provided by formal methods and the actual system through two case studies on embedded security architectures [7]. In the hardware verification context, mutation testing is a more popular way to gauge the incompleteness gap, as will be described in §2.4,

2.4. Mutation testing

Although FAST draws inspiration from mutation testing on hardware SPEC completeness, the idea of mutation testing actually originated from the skepticism on the correctness of software test suites. While the correctness of CODE is guarded by tests, there is nothing to ensure that the test suite itself is comprehensive enough. This is similar to the SPEC incompleteness problem FAST aims to solve.

In a high-level description, mutation testing assesses the quality of a test suite by applying mutations to a program and checking if the test suite reacts differently with the original CODE vs the CODE mutant [95]. Since its inception in 1970s [9, 19, 38], mutation testing has been applied to various use cases, as summarized below:

Completeness evaluation of different styles of tests. CODE testing has multiple types/styles such as, unit testing [87], integration testing [17, 36], end-to-end testing [73], etc. These related works use mutation testing to check the quality of different types of test suites. The general evaluation process is to conduct random mutations on the CODE and check to see if the mutant can be killed by any test case in the test suite.

Another way to categorize the related works is by the programming language in which CODE is implemented. Mutation rules have been introduced into different languages to test the effectiveness of the test suites (with a focus on unit tests). Examples of language mutation include: C++ [18], Java [57], Ruby [55]. Mutation testing can be used to ensure the effectiveness of large applications developed with multiple languages as well, such as web applications [77] and Android applications [65]. In these works, the integration and end-to-end tests are usually the subject of evaluation.

Mutation testing has also been used on programs that do not have concrete execution semantics. For example, several work targets mutation testing on finite-state machines (FSMs) [27, 58] which are only tested via simulation. They use a comprehensive checklist to select the mutation points.

> FAST is similar to this line of research in terms of producing valid CODE mutants. But FAST differs from them not only in the evaluation target (i.e., SPEC vs tests) but also in the way how FAST produces surviving CODE mutants and checks the quality of a mutant.

Completeness evaluation of hardware SPEC. In the hardware verification context, mutation testing has been used to improve the completeness of hardware SPEC [30, 51, 86]. The general process of mutation testing in hardware verification is to inject specific functional transformations in circuit (i.e., the CODE) programmed in languages like VHDL or Verilog. These programs (CODE mutants), are syntactically correct but functionally incorrect. The mutants will be given to the verifier together with the SPEC to see whether the mutated implementation may still satisfy the SPEC.

> FAST shares the same goal with this line of research: finding gaps in SPEC. But FAST faces two more challenges: 1) judging whether a surviving CODE mutant signals an intentional gap or a blind spot, and 2) producing surviving mutants with a much larger domain of random mutations. None of these problems are solved in the related works.

Mutation directly on SPEC. The earlier research work on SPEC mutation [4, 45, 62, 83] considered CODE as a black-box, and mutate the SPEC in order to find out incompleteness in SPEC [10], There are some implementations on FSM-based SPEC [16, 76].

> FAST differs from this line of research in that FAST mutates CODE instead of SPEC. SPEC are generally more versatile than CODE. For example, SPEC can be declarative, imperative, state-machines, etc, while CODE are typically imperative with a common set of operators such as binary operators. Therefore, the surveyed works are applied in highly-specialized context while CODE mutation-based framework like FAST has a higher chance of being generalized to other formally verified systems.

Generic improvement. Last but not least, the final line of related work focuses on improving the mutation testing approach in general. For example, several works sought to reduce the cost of mutation testing by selecting a subset of mutants [11], applying selective mutations [60], or adopting heuristics and search-based [96] mutant generation.

> FAST is orthogonal to these lines of research while its results can be integrated into FAST when applicable. We leave some of the integration items as future work §7.

2.5. Evolution strategy in fuzzing

Fuzzing (also known as fuzz testing) is a software testing scheme that checks the correctness of a program by repeatedly generating random inputs and monitoring the program executions for defects [59]. As the input space of a program is (in most cases) too huge for an exhaustive enumeration, strategically generating inputs that may bear a higher chance of triggering a bug is crucial. For example, a program that takes a string of bytes as input (such as XML

file parsers) has a (virtually) infinitely large input space and there is no way to exhaustively enumerate it.

Mutation testing faces a similar problem. Even with a medium-sized project, the number of potential CODE points for mutation multiplied by the potential ways to mutate each CODE point produces an extremely large search space. Furthermore, compared with testing (i.e., concrete execution), formal verification (i.e., abstract and symbolic execution) is usually orders of magnitude slower. For example, the S2N test suite finishes in seconds while the verification takes tens of minutes. This slowness further limits the applicability of exhaustive enumerations to small codebases only.

In modern fuzzing research, one way to deal with the state exploration problem is to simulate the natural selection process, with a combination of *random mutation* and *survival of the fittest*. To be specific, in each mutation round, *random mutation* is used to add more chance in exploring more path that has not been explored. the *survival of the fittest* process effectively ranks different seeds based on the feedback (e.g., code or path coverage in most fuzzing work), and gives the seed with a higher ranking a better chance to generate inputs for future rounds of testing.

Feedback loop. Evolutionary fuzzers use feedback from each loop of fuzzing to discover over time the execution state space of the program. Among all the building blocks of a modern fuzzer (e.g., mutation rules, seed scheduling, feedback mechanisms), the metric that provides an objective evaluation on the seed quality is of paramount importance to the effectiveness of a fuzzer. For example, the pioneer work American Fuzzy Lop [97] is an evolutionary fuzzer which uses code coverage to guide the process of seed generation. It maintains a seed queue that stores all the seeds, including the initial seeds chosen by the user as well as the ones that are mutated from the existing seeds and cause the program to reach new and unique execution states. This has inspired a fleet of coverage-guided works [8, 32, 56, 78, 81, 88].

Similar to evolutionary fuzzers, FAST navigates itself in a huge CODE mutant search space via a feedback mechanism. However, existing CODE-coverage based feedback is neither applicable in FAST nor can be easily exposed from the backend solvers. FAST proposes its new metric to evaluate a CODE mutant: the number and variety of verification errors from the solver.

Language fuzzing. Fuzzing has been used on different types of software systems as summarized by this survey [59]. One line of research that is especially related to FAST is language fuzzing. Language fuzzing aims to find issues with compilers or interpreters (e.g., virtual machines or JIT engines). For example, Superion [91] is an AFL-based fuzzer to find the bugs in XML and JavaScript engines. There are also other works aiming at JavaScript engines [24, 75], and Java language [44, 47]. Research works aiming at optimizing the language fuzzing process has been proposed as well. For example, generating the input mutant in a more efficient way [21, 26], some other works use different feedback patterns (e.g. code coverage) to guide the fuzzing mutant[41, 72].

FAST is similar to this line of research as they share the same target to mutate—CODE. However, existing works have not taken the completeness problem of SPEC into consideration. A smaller difference is that not all language fuzzing tools need to produce valid and type-checked CODE mutants, but this is a requirement for FAST.

3. The Tale of SPEC, CODE, and Tests

As demonstrated in the language fuzzing work (§2.5), creating a diverse set of CODE mutants is not a challenge for mutation testing. A more fundamental challenge is how to judge whether a surviving mutant is "meaningful". To be specific, in the context of FAST, when a CODE mutant passes the verification and signals a gap (e.g., Figure 2a), **how can we tell that the gap in the SPEC is by intention or by mistake?**

The insight behind FAST is to exploit and synergize the "redundancy" and "diversity" in formally verified programs: SPEC (from the spec team), CODE (from the dev team), and test suites (from the QA team) are all derived from the same set of requirements but programmed with totally different mentalities. It is therefore unlikely that the three teams (spec, dev, and QA) will make the same mistake. This paves the way for finding errors in one component by cross-comparing it against the other two. In fact, this principle is already applied to check the correctness of CODE by both running it against test suites and proving it against the SPEC. In this section, we show that the same procedures can be used to check incompleteness in SPEC as well (and deficiencies in test suites too, as by-products of our methods).

Notations. To precisely describe how FAST solves the problem, we first introduce some basic notations:

- We denote the SPEC to check as S, the CODE from devs as C, and the tests from QA as T.
- We denote the *refines-to* relation as $\sqsupseteq$. By definition, $S \sqsupseteq C$ as C verifies under S.
- We denote *semantically equivalence* as $\equiv$. $C \equiv C'$ means C behaves like C' in every observable way.
- Each test case $t \in T$ is a concrete input for C and C passes the test suite T, denoted as $C \succ T$, if and only if C passes every test case t.

Definition of a gap in SPEC. With these notations, we can formally define what a "gap" stands for in SPEC:

- Suppose we are able to hire an independent team of developers to work on the CODE and this new dev team produces new code C' where $(C' \not\equiv C) \wedge (S \sqsupseteq C \wedge S \sqsupseteq C')$. Then the semantic difference between C and C' (denoted as Δ_C) indicates a gap in the SPEC.
- Symmetrically [50, 51], gaps in SPEC can be exposed with an "alternative" spec team. Suppose we are able to find another SPEC S' such that $(S' \not\sqsupseteq S) \wedge (S \sqsupseteq C \wedge S' \sqsupseteq C)$. Then the difference between S and S' (denoted as Δ_S) represents a gap in the SPEC.

In the running example of Figure 1 and 2, C, S, and C' are shown in Figure 1a, 1b, and 2a, respectively. It is easy to observe that $(C' \not\equiv C) \wedge (S \sqsupseteq C \wedge S \sqsupseteq C')$, and this signals a gap in S.

Although it is possible to obtain C' or S' by hiring independent teams and to gauge Δ_C and Δ_S with expert reviews, such a practice is neither cost-effective nor scalable. This is where FAST fits into the picture. The mutation testing component of FAST plays the roles of independent dev and spec teams that produce C' and S'; while the gauging component of FAST judges whether a Δ_C or Δ_S signals a blind spot in the SPEC. In this paper, we focus on mutation testing to create Δ_C. More discussion on SPEC mutation can be found in §7.

Definition of a meaningful gap in SPEC. With the definition of a gap, how can FAST tell that the gap is *inadvertently* introduced into the SPEC? On first thought, this seems to be an unsolvable problem as SPEC are, by design, more abstract than CODE. To illustrate, assume SPEC S requires sorting the input but does not dictate a sorting algorithm. Therefore, the CODE is free to use either quick sort (C) or merge sort (C') to satisfy S, i.e., $S \sqsupseteq C \wedge S \sqsupseteq C'$ signals a gap in S. However, if gaps are indeed expected between SPEC and CODE, how can we tell that a gap is by intention or by mistake?

The solution is to invite the test suites (T) as an independent "referee" to the "rally" between C and S. To illustrate, in Figure 1, it is reasonable to expect that the `add1` function will be accompanied by a unit test $t \in T$ like the following:

```
assert add1([0,1,2]) == [1,2,3];
```

While this test t is unlikely to be written with the intention to block the code mutant C', C' will not pass this test case. In formal notations, we have $S \sqsupseteq C' \wedge C' \not\succ T$. In other words, the test suite (T) captures some valid intention that is not captured in the SPEC (S)—a strong indication that the gap in the SPEC is not intentional but more like a mistake.

Summary. There are only five possibilities after FAST obtains a CODE mutant and runs it for testing and proving:

① $S \sqsupseteq C' \wedge C' \succ T \wedge T \not\perp \Delta_C \implies$ there is a gap in the SPEC and this gap is intentional, as the test suites also explicitly allow this behavior (by the clause $T \not\perp \Delta_C$).

② $S \sqsupseteq C' \wedge C' \succ T \wedge T \perp \Delta_C \implies$ there is a gap in the SPEC and we cannot conclude whether the gap is intentional or mistaken, as the test suites also fail to capture this behavior (by the clause $T \perp \Delta_C$).

③ $S \not\sqsupseteq C' \wedge C' \succ T \implies$ the mutant is killed by the SPEC but passes the test suite, indicating that there might be incompleteness in the test suite.

④ $S \sqsupseteq C' \wedge C' \not\succ T \implies$ there is a gap in the SPEC and this gap is mistaken as it misses an important property that is captured even in concrete test cases.

⑤ $S \not\sqsupseteq C' \wedge C' \not\succ T \implies$ the mutant is killed by the SPEC and does not pass the test suite, this is expected and does not raise a signal.

When a gap is exposed through CODE mutants, FAST is able to infer the intention with the help of a robust test suite. If FAST is unable to deduce the intention for a particular gap, the gap represents some vacancy in the program semantics where neither the SPEC nor test suites cover. In such a case, FAST will report the gap to the users for manual analysis.

Level of manual effort When applied to an *automated* deductive verification system, FAST can find SPEC gaps automatically (case ④) while optional manual effort can help uncover more insights on the result (in other cases).

- Case ① and ②: manual checking can confirm whether $S \sqsupseteq C'$ is caused by an equivalent mutant, an intentional gap in SPEC, or incompleteness in both SPEC and tests.
- Case ③: manual checking can confirm whether $S \not\sqsupseteq C'$ is caused by out-of-sync proof hints (e.g., loop invariants) or a genuine SPEC violation. The latter case signals incompleteness in test suite, not SPEC.
- Case ④: requires no manual effort to confirm a SPEC gap.
- Case ⑤: $C' \not\succ T$ confirms C' is not an equivalent mutant. Manual checking can help decide whether $S \not\sqsupseteq C'$ is caused by missing manual proof hints (e.g., loop invariants) or a genuine SPEC violation. The former case might hide a gap in SPEC — this is a limitation of FAST.

4. Enumerative CODE Mutant Generation

With the SPEC gap classification problem solved in §3, the next road blocker of porting mutation testing into the software verification context is CODE mutant generation, i.e., **how to produce a CODE mutant that may pass the verification under the original SPEC**. In this section, we describe an enumerative strategy which is more suitable for small and simple codebases but is already effective enough to find shallow gaps in SPEC even for mature codebases. We describe a more sophisticated CODE mutation generation strategy which is more suitable for large and complex codebases in §5. It is important to emphasize here that in both strategies, when verifying the mutated code against SPEC, FAST will check the overall verification result instead of whether the modified function passes verification or not.

Type-preserving mutation. Recall that the goal of mutation is to produce *valid* CODE mutants that should at least compile and execute, otherwise, FAST won't be able to even verify and test the CODE mutant. This requires that whatever mutation rule FAST applies to convert C to C', the rule must respect the type system in which C is constructed. As a result, in FAST, all mutation rules are type-preserving by design. Table 1 show the list of mutation rules available in FAST that are considered type-preserving in most programming languages.

It is worth-noting that while the mutation rules in FAST preserve typing information, FAST does not guarantee that the CODE mutant must be semantically different from the original CODE, i.e., $C' \not\equiv C$. For example, `(a - a) * 2` will always evaluate to zero regardless of which rule we use to mutate the constant 2. However, in practice, such cases are extremely rare (and are most likely to be eliminated by compiler optimizations). The chances of producing a semantically equivalent CODE mutant under these rules are small and can be left to manual review *after* FAST have found a surviving mutant. We observed one such case in our experiments and presented it as a false positive case in §4.1.

Enumerative algorithm. Given the limited set of mutation rules discussed in Table 1, for small codebases that do not

#	Category	Mutation point		Mutate into
1	Unary	Neg	-	Drop the operator
2		Not	!	Drop the operator
3	Binary	Add	+	One of -, *, /, %
4		Sub	-	One of +, *, /, %
5		Mul	*	One of +, -, /, %
6		Div	/	One of +, -, *, %
7		Mod	%	One of +, -, *, /
8	Bitwise	BitAnd	&	One of |, ^
9		BitOr	|	One of &, ^
10		BitXor	^	One of &, |
11		Shl	«	One of $»_L$, $»_A$
12		LShr	$»_L$	Shl «
13		AShr	$»_A$	Shl «
14	Compare	Lt	<	One of <=, >=, >, ==, !=
15		Le	<=	One of <, >=, >, ==, !=
16		Ge	>=	One of <, <=, >, ==, !=
17		Gt	>	One of <, <=, >=, ==, !=
18	Equality	Eq	==	!=
19		Neq	!=	==
20	Constant	<value>		One of 0, 1, -1, MIN, MAX, etc.
21		<value>		One of value+1, value-1, etc.
22		<value>		A random value in range
23	Structure	<if-else>		Swap the branches
24		<continue>		break the loop
25		<break>		continue the loop
26		ITE	?:	Swap the operands

TABLE 1: Generic CODE mutation rules available in FAST

have many instructions in the original CODE, it might seem feasible to even try all possible mutation strategies using an algorithm described in algorithm 1.

Algorithm 1: Enumerative mutation testing

Input: Original CODE C, SPEC S, and test suite T
foreach Instruction $I \in C$ **do**
  if I has a mutation point **then**
    foreach rule r to mutate I **do**
      $\Delta_C \leftarrow \text{apply}(r, I)$;
      $C' \leftarrow \text{repackage}(\Delta_C, C)$;
      Run C' through verification and testing ;
      Check which of the following applies:
        1) $S \sqsupseteq C' \wedge C' \succ T \wedge T \not\perp \Delta_C$
        2) $S \sqsupseteq C' \wedge C' \succ T \wedge T \perp \Delta_C$
        3) $S \not\sqsupseteq C' \wedge C' \succ T$
        4) $S \sqsupseteq C' \wedge C' \not\succ T$
        5) $S \not\sqsupseteq C' \wedge C' \not\succ T$
      Report cases 2 and 4
    end
  end
end

An exponential search space. Note that algorithm 1 can be trivially extended to support the mutation of multiple CODE locations at the same time, i.e., producing high-order CODE mutant by mutating more than one instructions in the original CODE. Essentially, this means that with an enumerative approach, the search space is exponential to the number of mutable locations in the CODE. We denote the number of possible mutation locations in the CODE as n. For one instruction, there can be multiple possible mutation locations: operator, operand(s). Therefore, the search space will be 2^n where $n \geq I, I \in C$. However, it is still debatable on the effectiveness of high-order CODE mutants due to the *coupling effect* (more details in §6). FAST found mixed evidence on coupling effect in our case studies.

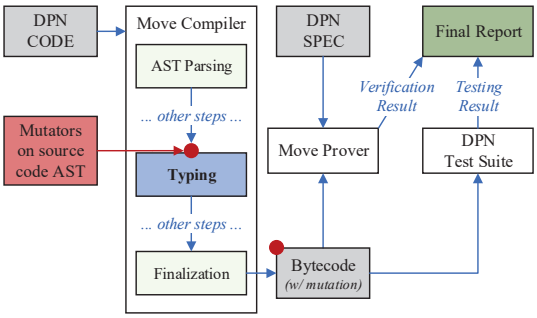

Figure 3: CODE mutation pass in the Move Prover pipeline

4.1. Case study: Diem Payment Network

Being a small yet critical smart contract, the Diem Payment Network [2] is a perfect case study for FAST to apply enumerative mutant generation strategy for finding blind spots in its comprehensive SPEC system.

Applying FAST to DPN. Figure 3 shows how FAST is applied to find SPEC blind spots on DPN. Briefly,

1) FAST first collects all possible mutation points in the DPN core CODE by statically analyzing the Move source code abstract syntax trees (ASTs) which are available in the Move compilation pipeline.
2) FAST then iteratively goes over each mutation point and follow the generic mutation rules in Table 1 to produce CODE mutant. For constant mutations, CODE randomly picks one of the three mutation rules listed in Table 1 to get the mutation target. All CODE mutations are inserted before typing step in the compiler to ensure that the CODE mutant is indeed valid Move code which in theory, can be developed by human developers.
3) With each CODE mutant generated, FAST passes it to the prover along with the original SPEC and check for verification results from the Move prover. The Move Prover will report the verification status and a detailed explanation of verification errors, i.e., which SPEC property is violated on which line of CODE, if any.

Findings. FAST identified 404 CODE locations where mutations can be applied—a number suitable for brute-force enumeration. The true omissions are summarized in Table 2, which is obtained by the following procedure:

- After enumerating each of the 404 CODE locations with one random mutation, together with higher-order mutations with a boundary of the number of mutation points used in constructing higher-order mutant. The boundary is set to 3 here. FAST reported 16 cases where the CODE mutant survived the verification.

#	File Name	Function Name	Mutation Point	Mutation Rule	Test	ℂ	§3	Details
1	DiemAccount.move	epilogue_common	Constant	1	Fail	✓	④	Case 1
2	DiemAccount.move	make_account	Constant	u16::MAX	Fail*	✓	④	
3	DualAttestation.move	initialize	Constant	+= 1	Fail*	✓	④	
4	AccountLimits.move	publish_window	Constant	+= 1	Pass	×	②	
5	AccountLimits.move	current_time	Constant	1	Pass	×	②	
6	CSRN.move	force_expire	Add	Sub	Fail	✓	④	
7	CSRN.move	shift_window_right	Constant	+= 1	Pass	×	②	
8	Diem.move	register_currency	Constant	0	Pass	×	②	Case 2
9	DiemConfig.move	emit_genesis_reconfiguration_event	Constant	+= 1	Fail*	✓	④	
10	DiemSystem.move	initialize_validator_set	Constant	1	Fail	✓	④	
11	DiemTimestamp.move	set_time_has_started	Constant	/= 2	Fail	✓	④	
12	SlidingNonce.move	publish	Constant	+= 1	Pass	×	②	
13	XUS.move	initialize	Constant	*= 2	Fail*	✓	④	
14	AccountLimits.move	can_withdraw_and_update_window	Ge	Gt	Pass	×	②	Case 3
15	DiemAccount.move	writeset_epilogue	Constant	1	Pass	✓	①	Case 4
16	DiemAccount.move	writeset_epilogue	Constant	1	Pass	✓	①	Case 4

TABLE 2: Findings on the DPN case study. Issues 1-13 are reported and fixed while 14-16 are confirmed to have no harm.

- Among the 16 surviving CODE mutants, 8 mutants failed the tests, including 4 mutants that passed the tests in its original setup but failed after an automated re-genesis-and-test infrastructure was later landed in the codebase (marked as "Fail*" in Table 2). 8 mutants passed the tests unconditionally, out of which 3 are covered by test cases.
- After analyzing all 16 cases, we confirmed 13 cases to be true omissions with SPEC fixed in pull request 1, 2, 3. The remaining 3 are false positives (explained later).

Sample reports. We present two true omissions, the false positive case, and the intended gap for readers' information.

Case 1: SPEC *omission signaled by a test failure.* In the following snippet, the original code will add the `sequence_number` with 1 at the end of this function. FAST mutated the constant `1` to be `0` and observed that the CODE mutant still passed the verification.

```
// code snippet in DiemAccount.move
fun epilogue_common<Token>(account: &signer)
acquires DiemAccount {
  let sender = Signer::address_of(account);
  let sender_account =
    borrow_global_mut<DiemAccount>(sender);
  sender_account.sequence_number =
    sender_account.sequence_number + 1;
    //!                            ^
    //! mut:                      1 -> 0;
}
```

However, the unit test failed because increasing the sequence number in the users' account is monitored by the following snippet of code in the unit test:

```
assert_eq!(sender_seq_num + 1, updated_sender.sequence_number());
```

In other words, this case obeys the pattern $S \sqsupseteq C' \wedge C' \not\succ T$, which is a clear signal that some intention failed to be captured in the SPEC. In fact, this is a serious loophole. It is a security requirement to increment the sequence number in the user's account after each transaction is sent, otherwise, the account can be vulnerable to replay attacks! The fix for this loop is to add an extra `ensures` clause in the SPEC, as shown below:

```
spec epilogue_common{
  //... redacted ...
  //! fix: added missing ensures
  ensures
    global<DiemAccount>(account).sequence_number
      == old(global<DiemAccount>(account).sequence_number) + 1;
}
```

Case 2: SPEC *omission confirmed manually.* In the code snippet below, FAST mutated the parameter that controls the exchange rate to the Diem coin when registering USD coin and yet this mutant managed to pass both verification and testing. This was a surprise as the stability of Diem coin is a core business requirement. However, later we noticed that the exchange rate is not used in DPN due to historical reasons.

```
// code snippet in XUS.move
fun initialize(dr_account: &signer, tc_account: &signer) {
  Diem::register_SCS_currency<XUS>(...,
    /* exchange rate = 1:1 */ FixedPoint32::new(1, 1)
    //!                                          ^
    //! mut:                                    ^ 1 -> 0;
  )
   // ... redacted ...
}
```

The fix is to add an extra `ensures` clause in the USD coin registration function.

Case 3: false positive due to semantic equivalence. While applying mutation rules in Table 1 will likely distort the semantics of the CODE, there might still be a small chance that a semantically equivalent CODE mutant can be produced, as shown in the following snippet.

```
// code snippet in AccountLimits.move
fun can_withdraw_and_update_window<CoinType>(
    amount: u64,
    sending: &mut Window<CoinType>,
) {
  // ... redacted ...
  sending.tracked_balance =
    if (amount >= sending.traced_balance) { 0 }
    //!        ^^
    //! mut:   >= -> > (i.e., greater than)
    else { sending.tracked_balance - amount };
  // ... redacted ...
}
```

Although the mutant (with >= mutated into >) passed the verification, it does not imply a gap in the SPEC. In fact, the mutant is semantically equivalent to the original CODE: when `amount == sending.tracked_balance`, the difference between them is `0`, therefore, it does not matter whether the difference is calculated in the `then` or `else` branch.

Case 4: intended gap in SPEC. Our manual analysis also revealed an intended gap in the SPEC, as shown below:

```
// code snippet of DiemAccount.move
fun writeset_epilogue(account: signer, sequence_number: u64) {
  epilogue_common<XUS>(account, sequence_number, 0, 0, 0);
  //!                                              ^  ^
  //! mut 1:                                       0 -> 1
  //! mut 2:                                          0 -> 1
}
fun epilogue_common<Token>(
  account: &signer, sequence_number: u64,
  gas_price: u64, max_gas_units: u64, gas_units_remaining: u64
) {
  let fee_amount = gas_price * max_gas_units;
  if (fee_amount > 0) {
    // ... redacted ...
    assert!(/* some condition P to abort */);
  }
}

spec epilogue_common{
  // ... redacted ...
  aborts_if
    (gas_price * max_gas_units > 0) && (/* some condition P */)
}
```

Notice the two parameters (`gas_price` and `max_gas_units`) in the parameter list of function `prologue_common`. Both the test and SPEC require the product of the two parameters to be `0`, but there are no specific requirements for the two parameters separately. As a result, mutating either of them from `0` to `1` has no effect on both testing and verification. This example (which maps to two reports by FAST) follows the pattern $S \sqsupseteq C' \wedge C' \succ T \wedge T \not\perp \Delta_C$. As a result, although both CODE mutants pass verification, they are considered to be an intended gap in SPEC.

5. Evolutionary CODE Mutant Generation

As shown in §4.1, the enumerative CODE mutant generation strategy works well for small codebases, however, when facing a larger codebase, enumerating all the possible CODE mutants is not a preferable approach as the number of possible mutants grow exponentially with the size of the codebase. Therefore, we need a strategy that can produce CODE mutants that are inherently more likely to pass the verification than random guessing.

To navigate the search space for surviving CODE mutant, FAST incorporates an evolutionary process in mutant generation, inspired by the effectiveness of coverage-guided fuzzers. In conventional fuzzing, unexpected inputs are fed to a program with the hope of triggering unsafe behaviors in the program. In FAST, the "unexpected" inputs are CODE mutants C' and "unsafe" behavior is defined when C' passes verification, i.e., $S \sqsupseteq C'$. Although C' passing the test suite is also "unsafe" (as they signal gaps in the test suite), they are by-products and the focus of the mutator is still to produce CODE mutants that pass the verification.

Like every genetic algorithm, FAST needs to answer two questions in its design: ① what to mutate in one evolution round and ② which mutant "fits" the environment and thus, should be given more opportunities to generate future seeds.

Mutation points. The solution to ① is to pre-collect potential mutation points in the CODE before evolution starts. In this information collection step, FAST scans the given CODE from beginning to end and matches every instruction with the possible mutation patterns defined in Table 1. Similar to the enumerative approach (§4), the mutation rules must preserve typing information after the transformation.

"Fitness" evaluation. The solution to ② is SPEC *coverage*, a simple metric to measure how far the mutant is from its evolution goal—passing the verification. In FAST, SPEC coverage is measured by the verification errors triggered by a CODE mutant. For each CODE mutant that fails the verification, FAST expects a report from the verifier to describe the failure. The report can be as simple as a binary pass/fail signal or a list of tuples (X, Y, Z) each contains a record on SPEC X fails on CODE location Y due to reason Z. Of course, the more verbose the information, the better it is for FAST to measure the "fitness" of a mutant. Fortunately, in practice, most formal verification tools can give a very detailed explanation of a verification error, some even include counterexamples that can be concretely executed to pinpoint the error.

Intuitively, CODE mutants that reduce verification errors reported in the "parent" mutant (i.e., a strict subset of errors) will be considered as "fit" and should be used to seed more mutants. Similarly, CODE mutants that uncover previously unknown verification errors are considered as increasing SPEC coverage, and hence, will be given more chances to mutate because this opens more diversity for evolution.

Each CODE mutant that is considered "fit" is assigned an initial score which is inversely related to two factors: 1) how many verification errors remain and 2) how long is the mutation trace. For the same set of verification errors, FAST favors the smaller mutant (i.e., smaller edit distance from the original CODE).

Seed scheduling. While the "fitness" evaluation decides whether a new CODE mutant should be considered as a seed for future rounds of mutation, FAST also needs to adjust the scores of the parent seed of this mutant. In general, FAST will reward the parent seed if the new mutant is "fitting" and penalize the parent seed if the new mutant is "boring". A mutant is "boring" if it neither expands the SPEC coverage nor fixes any verification error in the parent seed.

Overall fuzzing process. Figure 4 shows the evolutionary CODE mutant generation strategy in FAST. FAST maintains a *seed pool* to keep track of seeds that can be used for future mutation rounds. All seeds in the seed pool are ordered by their score. Each evolution round starts with the seed selection process which is essentially temporarily popping the seed with the highest score out of the seed pool. Then, an additional mutation step is applied to the selected seed and the new CODE mutant is sent for verification and testing.

- If the verification passes, depending on the results from the test suites, FAST will signal whether this CODE mutant signals an intentional or unintentional gap in the SPEC (or mark it as an inconclusive case).
- If the verification fails, FAST evaluate the "fitness" of the new CODE mutant and save a new seed if it "fit". FAST will also update the score of the parent seed and put it back into the seed pool as well.

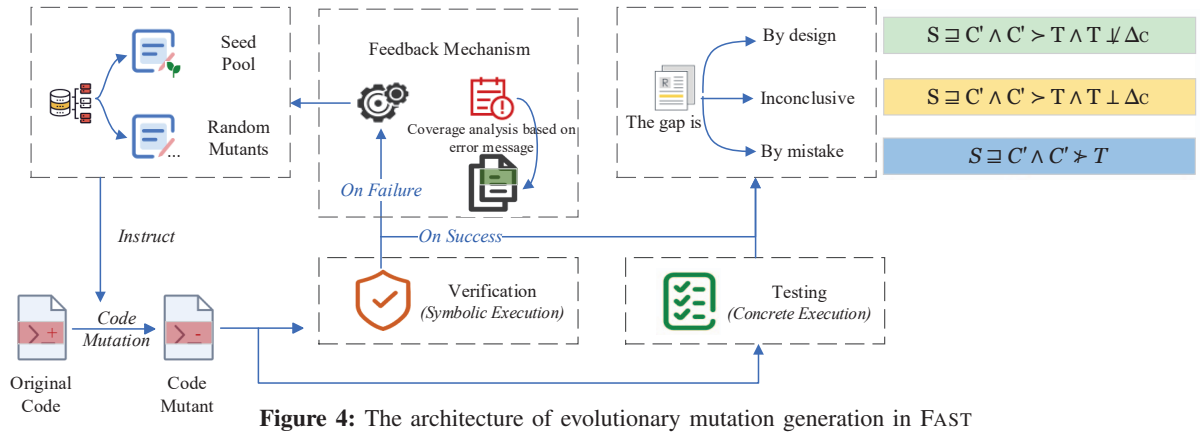

Figure 4: The architecture of evolutionary mutation generation in FAST

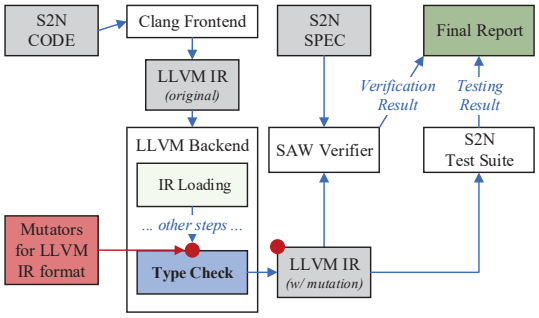

Figure 5: S2N Mutation Architecture

It is worth mentioning that unlike conventional fuzzing which can be jump-started from a seed pool with many test cases, at the very beginning, the seed pool in FAST has one seed only, which is the original CODE without any mutations. FAST gives this genesis seed a sufficiently high score to quickly populate a large number of single-mutation seeds in the pool. But after the bootstrapping period, this genesis seed is no different from other seeds in FAST's point of view.

5.1. Case study: AWS TLS implementation

An ideal showcase for evolutionary mutation testing is a codebase that is sophisticated enough (such that enumeration of mutations is not feasible) and yet extensively specified (such that gaps in SPEC are relatively rare). S2N, Amazon's home-grown TLS implementation, is a good candidate.

Applying FAST to S2N. While Figure 4 shows the overall fuzzing process implemented in FAST, Figure 5 shows how FAST is applied to find SPEC blind spots on S2N from the point of view of a single fuzzing round. Briefly,

1) FAST first collects all possible mutation points in S2N CODE. While it is doable at the C AST level, FAST chooses to statically analyze the LLVM IR which is obtained by compiling and linking together all relevant C source code. This is primarily for convenience reasons.
2) However instead of iteratively going over all mutation points to produce CODE mutant, FAST adopts the evolutionary scheme described in Figure 4 by rewarding CODE mutants that are more likely to succeed (i.e., pass the verification) in future rounds of mutations.
3) With each CODE mutant generated, FAST passes it to SAW (the verifier) along with the original SPEC and check for verification results. SAW will report the verification status and a detailed explanation of verification errors, i.e., which SPEC property in SPEC is violated and its reason.

Findings. FAST identified 6772 CODE locations where mutations can be applied—making brute-force enumeration infeasible, especially consider the possibilities of *high-order mutants*. Therefore, the best way to explore the search space is via evolutionary mutation. In particular, FAST starts with an empty seed (i.e., the original CODE) in the seed pool and on each fuzzing round, it chooses whether to replace the mutation on one CODE location (denoted as retrial mutants) or append a mutation to a new CODE location (i.e., creating high-order mutants). We ran FAST for 72 hours, we got a total of 348 surviving CODE mutants that passed the verification, out of which 12 are retrial mutants, 9 are high-order mutants, and the majority (327) are mutants obtained by applying mutation on a single CODE location in one trial.

Among these surviving mutants, we manually sampled 22 for initial analysis, with a prioritization on mutants that caused test case failures. Out of the 22 cases, 15 triggered test failures and we confirm that they all signal a loophole in the SPEC. The remaining 7 CODE mutants passed test suite, out of which 6 have no coverage on the mutation point and the one with coverage is confirmed to be an intended gap in the SPEC (with details explained later). These findings are summarized in Table 3. We have reported all findings to the development team of S2N with acknowledgment and are currently waiting for their patching (see our initial reporting for more samples other than the case studies shown here).

Sample reports. A surviving CODE mutant must be in one of the following categories: 1) one mutation trial on a single CODE location, 2) multiple mutation trials on a single CODE location, and 3) mutations on multiple CODE locations. We showcase a sample in each category as well as provide a detailed explanation for the intended gap FAST found.

#	File Name	Function Name	Mutation Point	Mutation Rule	Test	ℂ	§3	Details
1	s2n_drbg.c	s2n_drbg_bits	Sub	Add	Fail	✓	④	
2	s2n_handshake_io.c	s2n_validate_ems_status	Constant	1	Fail	✓	④	
3	s2n_socket.c	s2n_socket_write_uncork	Constant	-= 1	Pass	×	②	
4	s2n_handshake_io.c	s2n_generate_new_client_session_id	If-Else	Swap	Fail	✓	④	
5	s2n_drbg.c	s2n_drbg_mix	Add	Sub	Fail	✓	④	
6	s2n_handshake_io.c	s2n_conn_set_handshake_type	Not	Drop	Pass	×	②	Case 1
7	s2n_socket.c	s2n_socket_write_uncork	Eq	Neq	Fail	✓	④	
8	s2n_socket.c	s2n_socket_was_corked	Neq	Eq	Pass	×	②	
9	s2n_handshake_type.c	s2n_handshake_type_set_tls12_flag	BitOr	BitAnd → BitXor	Pass	×	②	Case 2
10	s2n_drbg.c	s2n_drbg_bits	Gt	Eq → Neq	Fail	✓	④	
11	s2n_drbg.c	s2n_drbg_instantiate	Lt	Ge → Eq → Le	Fail	✓	④	
12	s2n_drbg.c	s2n_drbg_mix_in_entropy	0	+=1 → set MAX → set -1	Fail	✓	④	
13	s2n_drbg.c	s2n_drbg_update	16	/=2 → +=1 → *=2	Fail	✓	④	
14	s2n_drbg.c	s2n_drbg_generate	49	/=3 → set MAX	Fail	✓	④	
15	s2n_drbg.c	s2n_drbg_seed	Gt	Le → Ge	Pass	×	②	
16	s2n_random.c	s2n_get_random_data	0	-=2 → set 2	Fail	✓	④	
17	s2n_blob.c	s2n_blob_zero	BitXor + If-Else	BitOr + Swap	Pass	×	②	Case 3
18	s2n_blob.c	s2n_blob_validate	Constant + BitAnd	set 1 + AShr	Fail	✓	④	
19	s2n_drbg.c	s2n_drbg_block_encrypt	Constant + If-Else	set -1 + Swap	Fail	✓	④	
20	s2n_drbg.c s2n_blob.c	s2n_increment_drbg_counter s2n_blob_validate	Add BitAnd	Shl LShr	Fail	✓	④	
21	s2n_fork_detection.c s2n_socket.c	s2n_get_fork_generation_number s2n_socket_was_corked	Constant BitAnd	/= 3 Add	Fail	✓	④	
22	s2n_handshake_io.c	s2n_conn_set_tls13_handshake_type	Constant	+=1	Pass	✓	①	Case 4

TABLE 3: The sampled results from the S2N case study. All reports have been submitted to the S2N development team.

Case 1: single location single mutation. In the following snippet, FAST obtained a surviving CODE mutant by negating one of the predicates in a `if` condition.

```
// a simplified code snippet in s2n_handshake_io.c
int s2n_conn_set_handshake_type(struct s2n_connection *conn) {
  // ...redacted...
  if ((
    conn->mode == S2N_SERVER &&
    conn->status_type == S2N_STATUS_REQUEST_OCSP &&
    conn->handshake_params.our_chain_and_key &&
#!  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#!  mut: negate this condition, i.e., replace it into below
#!  'conn->handshake_params.our_chain_and_key == NULL'
    conn->handshake_params.our_chain_and_key->ocsp_status.size > 0
  ) || s2n_server_sent_ocsp(conn)) {
    s2n_handshake_type_set_tls12_flag(conn, OCSP_STATUS));
  }
  return S2N_SUCCESS;
}
```

The mutant passes both verification and testing. After investigation, we found out that this gap is caused by the fact that the value `our_chain_and_key` is neither monitored in the SPEC nor in test. FAST therefore identified it as a $S \sqsupseteq C' \wedge C' \not\succ T$ case. A closer examination of the code revealed that this is a dangerous modification: in the CODE mutant, should execution ever reaches this `if`-statement, it is guaranteed that the program will crash due to a `null`-pointer dereference. This at least violates one of the high-level guarantees that there should be no memory errors in the S2N codebase and yet neither test nor SPEC covers it.

Case 2: single location with multiple mutation trials. In the snippet below, We observed that FAST first attempted to mutate operator `|=` to `&=`. Although this attempt failed, FAST was able to discover new verification errors which allows the seed to have further mutations. In the next round of mutation, FAST replaced `|=` with `^=`. This CODE mutant passed both verification and testing.

```
S2N_RESULT s2n_handshake_type_set_tls12_flag(
    struct s2n_connection *conn,
    s2n_tls12_handshake_type_flag flag)
{
    // ... redacted ...
    conn->handshake.handshake_type |= flag;
    #!                             ^^
    #! mut trial 1:                |= -> &=  // fail verification
    #! mut trial 2:                |= -> ^=  // pass verification
    // ... redacted ...
}
```

During investigation, we noticed that the SPEC attempts to confine the possible values of `handshake_type`, as shown below:

```
// a redacted spec for the function being verified
conn_set_pre_tls13_handshake_type : connection -> connection
conn_set_pre_tls13_handshake_type conn = conn'
  where conn' = {handshake = handshake', /* redacted */}
        (handshake' : handshake) = {
            handshake_type = handshake_type'
            /* redacted */
        }
        handshake_type' =
            NEGOTIATED || full_handshake ||
            perfect_forward_secrecy || ocsp_status || ...;
```

When the operator is mutated to `&=`, the result of `conn->handshake.handshake_type` can be `0` (e.g., when `conn->handshake.handshake_type == 0`). But `0` is not allowed by SPEC, hence the verification failure. However, when the operator is mutated to `^=`, all the possible results are included in the SPEC. interestingly, the counterpart function for TLS 1.3 does not suffer from this incompleteness issue.

Case 3: mutations on multiple CODE *locations*. In the snippet shown below, FAST applied two mutations on different CODE locations. The net effect of the two mutations

is essentially marking the precondition to be unconditionally true. The mutant passes both verification and testing. However, applying any single mutation led to verification failure.

```
int s2n_blob_zero(struct s2n_blob *b) {
  POSIX_PRECONDITION(s2n_blob_validate(b));
#!                    ^^^^^^^^^^^^^^^^^^^
#! mut (net effect)  s2n_blob_validate(b) -> TRUE
  POSIX_CHECKED_MEMSET(b->data, 0, MAX(b->allocated, b->size));
  POSIX_POSTCONDITION(s2n_blob_validate(b));
  return S2N_SUCCESS;
}

// with POSIX_PRECONDITION pre-unrolled.
int s2n_blob_zero(struct s2n_blob *b) {
  S2N_RESULT result = s2n_blob_validate(b);
  if (result ^ 1) {
#!           ^
#! mut 1:    ^ -> | (bit-or)            // fail verification
#! mut 2:    swap the if-else branches  // pass verification
    return S2N_FAILURE;
  } else {
    POSIX_CHECKED_MEMSET(b->data, 0, MAX(b->allocated, b->size));
    POSIX_POSTCONDITION(s2n_blob_validate(b));
    return S2N_SUCCESS;
  }
}
```

In fact, it is surprising that the verification can even pass without the precondition requiring the input `blob` to be valid.

Case 4: an intended gap. The intended gap can be illustrated with the following code snippet with mutation done by FAST inlined:

```
static S2N_RESULT
s2n_conn_set_tls13_handshake_type(struct s2n_connection *conn) {
    // ... redacted ...
    if (conn->psk_params.chosen_psk == NULL) {
        // The constant FULL_HANDSHAKE bears value 2
        s2n_handshake_type_set_flag(conn, FULL_HANDSHAKE);
        #!                                ^^^^^^^^^^^^^^
        #! mut:                          FULL_HANDSHAKE -> 3
        #! i.e. FULL_HANDSHAKE -> FULL_HANDSHAKE | NEGOTIATED
    }
    // ... redacted ...
    return S2N_RESULT_OK;
}
```

By mutating the constant `FULL_HANDSHAKE` (aliased to integer 2) to 3, the new CODE still passes the full suite of verification and tests. Upon further investigation, we notice the definition of the `s2n_handshake_type_flag` is an `enum` in C language:

```
typedef enum {
    INITIAL                         = 0,
    NEGOTIATED                      = 1,
    FULL_HANDSHAKE                  = 2,
    CLIENT_AUTH                     = 4,
    NO_CLIENT_CERT                  = 8,
} s2n_handshake_type_flag;
```

Hence, logically, after mutation, the new CODE is setting the flag to be `FULL_HANDSHAKE | NEGOTIATED`.

S2N indeed has a dedicated SPEC for this function (shown below) which *explicitly* allows the `NEGOTIATED` flag to be either set or unset. which explains why this is an intended gap explicitly allowed in the SPEC.

```
// a redacted spec  for the function being verified
conn_set_tls13_handshake_type : connection -> connection
conn_set_tls13_handshake_type conn = conn'
    where conn' = {handshake = handshake', /* redacted */ }
    (handshake' : handshake) = {
        handshake_type = handshake_type',
        /* redacted */
    }
    handshake_type'  = NEGOTIATED || full_handshake
                       || /* redacted */
    full_handshake   = if conn.chosen_psk_null
                       then FULL_HANDSHAKE
                       else 0
    // spec for other fields are redacted
```

6. Extra Evaluations

While the effectiveness and practicality of FAST is evaluated on the two real-world case studies (§4.1 and §5.1), in this section, we highlight some extra statistics that may help justify the key design choices of FAST.

Effectiveness of test suite. We evaluate the effectiveness of using test suite as a referee in categorizing a gap found in SPEC, i.e., whether the gap is by intention or by mistake. The evaluation is based on the mutants that successfully pass the verification in both codebases, and the test suites we used here are the unit tests, integration tests, as well as end-to-end tests available in the codebase.

Table 2 shows the overall result in DPN. For all 16 cases which we report, FAST is able to automatically judge whether a gap in the SPEC is intentional or mistaken in 10 cases (8 blind spots and 2 intentional gaps), showing a 62.5% automation rate on gap categorization. Table 3 shows the 22 cases we investigated in S2N-TLS. FAST is able to automatically categorize SPEC gaps in 16 of them, showing a 72.7% automation rate. Based on these results, it is reasonable to conclude that using test suite as a referee for SPEC incompleteness judgment is feasible.

Coupling effect. Coupling effect has a non-neglectable influence on the usefulness of producing higher-order mutant in FAST. Coupling effect came to researchers' notice shortly after the appearance of mutation testing [19]. The idea is that mutants can be limited to simple one-hop changes without impairing much on the overall effectiveness. This is because complex faults can be decoupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults. Coupling effect has got both empirical [70] and theoretical [89, 90] support.

Based on the result in the case studies, we do observe that coupling effect has an impact on the usefulness of generating high-order mutants in FAST. Given the relatively small size of the DPN codebase, we indeed attempted to enumerate all two and three-CODE-location mutants in the DPN but this exercise yielded no new findings, hence, making a strong indication that high-order mutants might be of limited value in small codebases. The S2N results are more encouraging: while the majority of surviving CODE mutants are still single-location mutations, we start to observe CODE mutants that must rely on two or more mutations to survive and usually signals a more interesting gap. We expect that coupling effect is stronger in small codebases while high-order mutations are more useful in medium to large codebases.

Effectiveness of evolution. Figure 6 shows the accumulated number of surviving mutants found by FAST as evolutionary mutation testing continues to run on S2N. All experiments are performed on a server running Ubuntu 20.04 with an Intel Xeon E7-8870 (2.40GHz CPU) with 80 cores and 1 TB RAM. Consistent with conventional fuzzing, in FAST, the

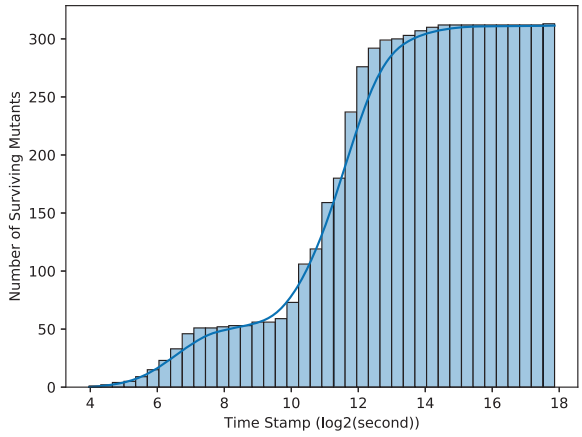

Figure 6: Seed Distribution

rate of producing new surviving CODE mutants decreases gradually over time until reaching a saturation point.

7. Discussion and Future Work

Mutating SPEC. FAST finds gaps in the SPEC by keeping the SPEC intact and mutating the CODE. Symmetrically, and also proved in theory [50, 51], we can find the same gaps by keeping the CODE unmodified and mutating the SPEC. In fact, as discussed in §2.4, the initial works on applying mutation testing into a formal verification context mainly focuses on mutating the expressions in the SPEC [10, 45, 71].

However, the symmetry of CODE and SPEC mutation only applies to finding gaps in the SPEC and does not apply to the process of judging whether the gap is intentional or mistaken. In FAST, we can categorize the gap by simply running the tests against the CODE mutant. But for SPEC mutants, running tests is futile as the CODE is unmodified. This is the primary reason why FAST does not adopt the SPEC mutation approach.

The solution to extend the symmetry into the gap categorization process is SPEC embedding, i.e., embedding SPEC at proper CODE locations, denoted as C_S. Given that C_S is executable, we can run any SPEC mutant $C_{S'}$ against the test suit T and check whether $C_{S'}$ passes T. Note that in this case, the gap in the SPEC is considered mistaken if $C_{S'} \succ T$ and intentional if the test fails. And yet, even with the symmetry extended, embedding logic is written in a more expressive language and has its own set of challenges (e.g., unrolling existential and universal quantifiers).

Coverage tooling for SPEC completeness It is worth noting that while mutation testing is initially proposed to gauge the completeness of the test suite, it is rarely considered a mainstream approach for this purpose. What is more popular now is various CODE coverage metrics, usually presented in terms of line coverage, instruction coverage, or branch coverage. Paranoid maintainers of open-source projects may even require that any new CODE needs to be accompanied by test cases to maintain a high ratio of CODE coverage in the codebase. As a result, the community has accumulated a sufficient set of tooling for CODE coverage measurement and reporting.

In a no-so-surprising contrast, to the best of our knowledge, there is no such tooling to measure CODE coverage for SPEC. It is not hard to imagine that such coverage tracking tools will be extremely challenging to build. Every CODE snippet seems to participate in the proving of some SPEC properties based on how the verification problems are handled in state-of-the-art verifiers [5, 12, 31, 52], and it is hard to untangle the complicated logical formula. However, despite the technical challenges, we believe that such tooling is necessary when formal verification gains enough traction and we hope that the findings from FAST can serve as a weak call to build coverage tracking tooling tailored for SPEC among the community.

The applicability of FAST FAST is applicable to a formally verified software when two conditions are met: 1) the verification system is fully automated, and 2) FAST can modify some form of CODE representation (e.g. LLVM IR). Therefore, besides the SAW toolchain, FAST is also compatible with combinations like LLVM + SeaHorn, LLVM + Kani (for Rust) etc. For adapting FAST to new verification systems, e.g., CBMC, a new mutator is required because CBMC has its own version of language IR.

The general applicability of FAST is limited at the moment, as formal verification is yet to be a standard industrial practice (unlike testing), hence the lack of SPEC components in most software. However, we believe formal methods will gain traction and now is the perfect time to build the necessary tools to warn about potential defects in SPEC before its too late.

Causes of missing SPEC gaps by FAST FAST cannot find all potential gaps in SPEC for at least two reasons: First, the mutant evolution approach is inherently incomplete. Similar to why fuzzing cannot find all bugs in a software, the evolutionary mutation strategy cannot produce CODE mutants that uncover all gaps in SPEC— the search space is too large to enumerate. Second, as discussed in §3, certain SPEC gaps require manual effort to confirm, especially in case ⑤ where the CODE mutant fails verification — manual effort is needed to check whether the verification failure is caused by out-of-sync proof hints (which hides a SPEC gap) or a genuine SPEC violation.

8. Conclusion

In this paper, we present FAST, a tool for exposing incompleteness issues in formal SPEC. FAST shows how the "redundancy" and "diversity" in formally verified programs (SPEC, CODE, and test suites) can be synergized for cross-checking and provides concrete designs and implementations for SPEC blind spots detection via enumerative and evolutionary mutation testing. We applied FAST to DPN and S2N and confirmed 13 and 21 blind spots in their SPEC respectively. This highlights the prevalence of SPEC incompleteness in real-world applications. We hope the findings from FAST can serve as a weak call to draw more attention on measuring and ensuring the quality of SPEC in formally verified codebases.

References

[1] Tesnim Abdellatif and Kei-Léo Brousmiche. Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models. In *Proceedings of the 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, Paris, France, February 2018.

[2] Diem Association. Diem. https://www.diem.com/, 2022.

[3] Vytautas Astrauskas, Aurel Bílỳ, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J Summers. The prusti project: Formal verification for rust. In *Proceedings of the 22th NASA Formal Methods Symposium (NFM)*, Pasadena, CA, May 2022.

[4] Emine G Aydal, Richard F Paige, Mark Utting, and Jim Woodcock. Putting formal specifications under the magnifying glass: Model-based testing for validation. In *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST)*, Denver, CO, April 2009.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A Modular Reusable Verifier for Object-oriented Programs. In *Proceedings of the 2005 International Symposium on Formal Methods for Components and Objects (FMCO)*, Amsterdam, The Netherlands, August 2005.

[6] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, 2019.

[7] Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[9] Tim Budd and Fred Sayward. Users guide to the pilot mutation system. *Yale University, New Haven, Connecticut, Technique Report*, 114, 1977.

[10] Timothy A Budd and Ajei S Gopal. Program testing by specification mutation. *Computer languages*, 10(1):63–73, 1985.

[11] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. Yale University, 1980.

[12] Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman, and Aaron Tomb. SAW: The Software Analysis Workbench. In *Proceedings of the 21st European symposium on programming (ESOP)*, Pittsburgh, PA, March 2013.

[13] Bharvi Chhaya, Shafagh Jafer, and Umut Durak. Formal verification of simulation scenarios in aviation scenario definition language (asdl). *Aerospace*, 5(1):10, 2018.

[14] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA-visual Automated Transformations for Formal Verification and Validation of UML Models. Washington, D.C., September 2017.

[15] de Moura, Leonardo and Bjørner, Nikolaj. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, March–April 2008.

[16] Simone Do Rocio Senger De Souza, Jose Carlos Maldonado, Sandra Camargo Pinto Ferraz Fabbri, and Wanderley Lopes De Souza. Mutation testing applied to estelle specifications. *Software Quality Journal*, 8(4):285–301, 1999.

[17] Marcio Eduardo Delamaro, Jose Carlos Maldonado, and Aditya P Mathur. Integration Testing Using Interface Mutation. In *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE)*, New York, NY, November 1996.

[18] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. Assessment of class mutation operators for c++ with the mucpp mutation system. *Information and Software Technology*, 81:169–184, 2017.

[19] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[20] Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. A tour of cvc4: How it works, and how to use it. In *Proceedings of the 2014 International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Lausanne, Switzerland, October 2014.

[21] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language Fuzzing Using Constraint Logic Programming. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Vasteras Sweden, September 2014.

[22] David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. Fast and Reliable Formal Verification of Smart Contracts with the Move Prover. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Munich, Germany, April 2022.

[23] Nicolas Dilley and Julien Lange. Automated Verification of Go Programs via Bounded Model Checking. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Melbourne, Austrailia, November 2021.

[24] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2021.

[25] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Proceedings of the 8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, Toronto, Canada, July 2016.

[26] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. Evolutionary Grammar-based Fuzzing. In *Proceedings of the 12th International Symposium on Search Based Software Engineering (SSBSE)*, Bari, Italy, October 2020.

[27] Sandra Camargo Pinto Ferraz Fabbri, Jose Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. Mutation Testing Applied to Validate Specifications Based on Statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE)*, Boca Raton, FL, November

1999.

[28] Nicole Fern and Kwang-Ting Cheng. Detecting Hardware Trojans in Unspecified Functionality Using Mutation Testing. In *Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Austin, TX, November 2015.

[29] Nicole Fern and Kwang-Ting Cheng. Mining Mutation Testing Simulation Ttraces for Security and Testbench Debugging. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Irvine, CA, November 2017.

[30] Nicole Fern and Kwang-Ting Cheng. Evaluating Assertion Set Completeness to Expose Hardware Trojans and Verification Blindspots. In *Proceedings of the 2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Florence, Italy, March 2019.

[31] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where Programs Meet Provers. In *Proceedings of the 22nd European symposium on programming (ESOP)*, Rome, Italy, March 2013.

[32] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, Boston, MA, August 2020.

[33] Mathias Fleury. Optimizing a Verified SAT Solver. In *Proceedings of the 11th NASA Formal Methods Symposium (NFM)*, Houston, TX, May 2019.

[34] Galois. What4: New library to help developers build verification and program analysis tools. https://github.com/GaloisInc/what4, 2022.

[35] Michael JC Gordon. Hol: A proof generating system for higher-order logic. In *VLSI specification, verification and synthesis*, pages 73–128. Springer, 1988.

[36] Mark Grechanik and Gurudev Devanla. Mutation Integration Testing. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Vienna, Austria, August 2016.

[37] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, July 2015.

[38] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE transactions on software engineering*, (4):279–290, 1977.

[39] John Harrison. Formal Verification at Intel. In *Proceedings of the 18thACM/IEEE Symposium on Logic in Computer Science (LICS)*, Ottawa, Canada, June 2003.

[40] Katharina Hofer-Schmitz and Branka Stojanović. Towards formal verification of iot protocols: A review. *Computer Networks*, 174:107233, 2020.

[41] Zhijian Huang and Yongjun Wang. Jdriver: Automatic driver cclass generation for afl-based java fuzzing tools. *Symmetry*, 10(10):460, 2018.

[42] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

[43] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. *NASA Formal Methods*, 6617:41–55, 2011.

[44] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jFuzz: A concolic Whitebox Fuzzer for Java. In *Proceedings of the 1st NASA Formal Methods Symposium (NFM)*, Moffett Field, CA, April 2009.

[45] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[46] Christoph Kern and Mark R Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[47] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[48] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[49] John C Knight. Software Challenges in Aviation Systems. In *Proceedings of the 21st International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, Catania, Italy, September 2002.

[50] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Portland, ON, November 2008.

[51] Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. On the Duality between Vacuity and Coverage. Technical Report UCB/EECS-2008-26, EECS Department, University of California, Berkeley, March 2008.

[52] K Rustan M Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, Dakar, Senegal, April 2010.

[53] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[54] Nancy Leveson. Completeness in Formal Specification Language Design for Process-control Systems. In *Proceedings of the 2000 Workshop on Formal Methods in Software Practice (FMSP)*, Portland, OR, August 2000.

[55] Nan Li, Michael West, Anthony Escalona, and Vinicius HS Durelli. Mutation Testing in Practice Using Ruby. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Graz, Austria, April 2015.

[56] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[57] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[58] José Carlos Maldonado, Márcio Eduardo Delamaro, San-

dra CPF Fabbri, Adenilso da Silva Simão, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation testing for the new century*, pages 113–116. Springer, 2001.

[59] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[60] Pedro Reales Mateo and Macario Polo Usaola. Reducing mutation costs through uncovered mutants. *Software Testing, Verification and Reliability*, 25(5-7):464–489, 2015.

[61] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[62] Tim Miller and Paul Strooper. A framework and tool support for the systematic testing of model-based specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(4):409–439, 2003.

[63] Felipe R Monteiro, Mikhail R Gadelha, and Lucas C Cordeiro. Model checking c++ programs. *Software Testing, Verification and Reliability*, 32(1):e1793, 2022.

[64] Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. Z3str4: A Multi-armed String Solver. In *Proceedings of the 24th International Symposium on Formal Methods (FM)*, Beijng, China, November 2021.

[65] Kevin Moran, Michele Tufano, Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Gabriele Bavota, Christopher Vendome, Massimiliano Di Penta, and Denys Poshyvanyk. Mdroid+: A Mutation Testing Framework for Android. In *Proceedings of the 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, Gothenburg, Sweden, May–June 2018.

[66] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: Do-178c alternatives and industrial experience. *IEEE software*, 30(3):50–57, 2013.

[67] Yvonne Murray and David A Anisi. Survey of Formal Verification Methods for Smart Contracts on Blockchain. In *Proceedings of the 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, Canary Island, Spain, February 2019.

[68] Lee Naish. Specification= Program+ Types. In *Proceedings of the 7th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Pune, India, December 1987.

[69] J Norhuzaimin and HH Maimun. The Design of High Speed UART. In *Proceedings of the 2005 IEEE Asia-Pacific Conference on Applied Electromagnetics (APACE)*, Johor Bahru, Malaysia, December 2005.

[70] A Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[71] Vadim Okun. *Specification Mutation for Test Generation and Analysis*. University of Maryland, Baltimore County, 2004.

[72] Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: Coverage-guided Property-based Testing in Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, July 2019.

[73] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[74] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. Generative type-aware mutation for testing smt solvers. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–19, 2021.

[75] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing Javascript Engines with Aspect-preserving Mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[76] Pinto Ferraz Fabbri, S.C. and Delamaro, M.E. and Maldonado, J.C. and Masiero, P.C. Mutation Analysis Testing for Finite State Machines. In *Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE)*, Monterey, CA, November 1994.

[77] Upsorn Praphamontripong and Jeff Offutt. Applying Mutation Testing to Web Applications. In *Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Paris, France, April 2010.

[78] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[79] Martin Schickel, Volker Nimbler, Martin Braun, and Hans Eveking. An efficient synthesis method for property-based design in formal verification: On consistency and completeness of property-sets. In *Advances in Design and Specification Languages for Embedded Systems*, pages 179–196. Springer, 2007.

[80] Steve Schmidt. Introducing s2n-tls, a new open source tls implementation. https://aws.amazon.com/blogs/security/introducing-s2n-a-new-open-source-tls-implementation/, 2022.

[81] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. {kAFL}:{Hardware-Assisted} Feedback Fuzzing for {OS} Kernels. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[82] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal Verification of Avionics Software Products. In *Proceedings of the 16th International Symposium on Formal Methods (FM)*, Eindhoven, Netherlands, November 2009.

[83] Tatiana Sugeta, José Carlos Maldonado, and W Eric Wong. Mutation testing applied to validate sdl specifications. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC)*, Oxford, UK, March 2004.

[84] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, Montreal, Canada, April 2019.

[85] Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 18(4):36–45, 2001.

[86] Patrice Vado, Yvon Savaria, Yannick Zoccarato, and Chantal Robach. A Methodology for Validating Digital Circuits with Mutation Testing. In *Proceedings of the 2000 IEEE*

International Symposium on Circuits and Systems (ISCAS), Geneva, Switzerland, May 2000.

[87] Auri Marcelo Rizzo Vincenzi, José Carlos Maldonado, Ellen Francine Barbosa, and Márcio Eduardo Delamaro. Unit and integration testing strategies for c programs using mutation-based criteria. In *Mutation testing for the new century*, pages 45–45. Springer, 2001.

[88] Dmitry Vyukov. Syzkaller, 2015.

[89] KS How Tai Wah. Fault coupling in finite bijective functions. *Software Testing, Verification and Reliability*, 5(1):3–47, 1995.

[90] KS How Tai Wah. A theoretical study of fault coupling. *Software testing, verification and reliability*, 10(1):3–45, 2000.

[91] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware Greybox Fuzzing. In *Proceedings of the 41th International Conference on Software Engineering (ICSE)*, Montreal, Canada, May 2019.

[92] Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The isabelle framework. In *International Conference on Theorem Proving in Higher Order Logics*, pages 33–38. Springer, 2008.

[93] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[94] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New YorkNY, June 2020.

[95] W Eric Wong. *Mutation Testing for the New Century*, volume 24. Springer Science & Business Media, 2001.

[96] Baowen Xu, Xiaoyuan Xie, Liang Shi, and Changhai Nie. Application of genetic algorithms in software testing. In *Advances in Machine Learning Applications in Software Engineering*, pages 287–317. IGI Global, 2007.

[97] Michal Zalewski. American fuzzy lop, 2017.

[98] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

Appendix

We provide more background information on the two case studies covered in §4.1 and §5.1 respectively as well as a discussion on how loop invariants might have an impact on FAST in automatically confirming gaps in SPEC.

1. Background on Diem Payment System

The Move programming language. Move [6] is a programming language developed for smart contracts by Meta although it has transitioned into a community-backed project now. The language features formal verification at its core through its home-grown verification tool Move Prover [22], which statically verifies the correctness of Move smart contracts modeled with the Move Specification Language.

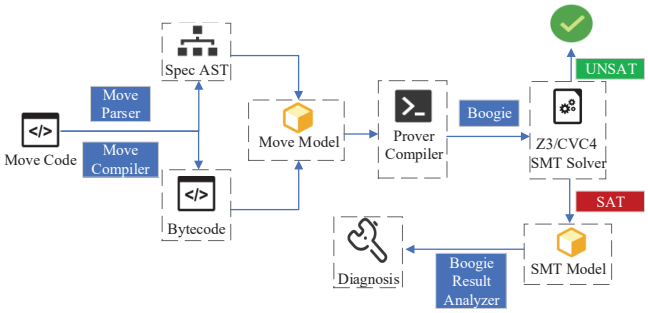

Figure 7: Move Prover architecture and workflow

The Move Prover. The architecture of the Move Prover is shown in Figure 7. Move CODE and SPEC are treated as input to the prover. The source code will be compiled into bytecode and the SPEC will be parsed into AST. Two parts will then go through a pipeline of merging and transformation and finally be compiled into Boogie [5], the intermediate verification language. The verification conditions in Boogie format will then be translated into SMT format which can be solved by SMT solver such as Z3 [15] or CVC4 [20].

The Diem Payment Network (DPN). DPN is the first major client of Move and Move Prover. The smart contract aims to function as a full-fledged and versatile payment/banking system with capabilities of handling multiple currencies, account roles, and rules for transactions. The DPN features a 7:5 CODE-SPEC ratio (with around 2,000 lines of core CODE in Move) which shows how the codebase is extensively specified. Most importantly, formal verification on DPN is *fully automated and runs continuously with unit and integration tests*, all open-sourced on GitHub—making DPN a perfect case study to test the effectiveness of FAST. The CI test coverage for DPN is 73%.

2. Background on AWS TLS Implementation

About S2N. Amazon S2N-TLS [80] is a C99 implementation of the TLS/SSL protocols. The previous de facto reference implementation contains more than 500,000 lines of code with at least 70,000 of those involved in processing TLS. In contrast, S2N implements the TLS protocol with less than 32,000 lines of code. Most of the implementations, including both the cryptographic primitives (e.g., HMAC) and the protocol itself (e.g., TLS handshake), are specified using SAW script [25]. The SAW toolchain is responsible for proving that the CODE conforms to the SPEC. The CI test code coverage rate for s2n-tls is 89.87%.

About SAW Software Analysis Workbench (SAW) is an industrial verification tool designed to prove equivalence properties between abstract SPEC and concrete CODE. The architecture of SAW is shown in Figure 8. It takes functions in LLVM IR as well as SAW-script to bridge the IR and the verification toolchain. If a function has an associated Cryptol SPEC, it will also be symbolically executed. The function terms and SPEC terms will be proven to be equivalent using What4 [34] behind the scenes.

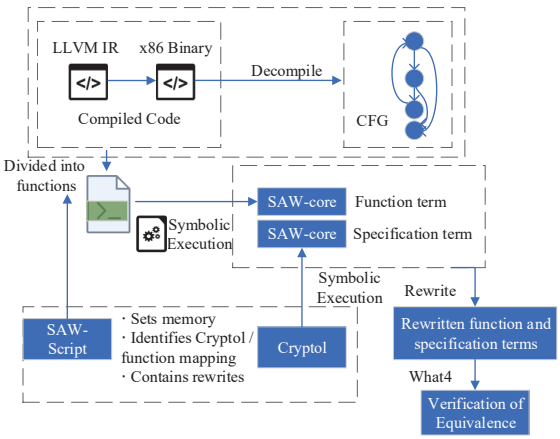

Figure 8: SAW architecture and workflow

3. Loop Invariants

FAST requires an automated deductive verification system to be the verifier and trusts its capacity in proving (or disproving) that an arbitrary CODE mutant conforms to the SPEC. In reality, deductive verifiers are often less capable than expected and might give up on solving complicated puzzles. A prominent example is proving post conditions for a function with loops in its control flow, as shown in Figure 9.

Instrumenting loop invariants is a typical approach to overcome this challenge. Effectively, loop invariants serve as hints to the automated prover and guides it to the proving of function postconditions. The drawback of adding loop invariants is that the proof hints are tightly coupled with the CODE and if the CODE changes, e.g., via mutation by FAST, the mutant C' might still satisfy the postconditions but the prover won't be able to draw the same conclusion due to out-of-sync loop invariants.

In other words, although C' appears to be $S \not\sqsupseteq C' \wedge C' \not\succ T$ (case ⑤ in §3), it might actually be case ④ $S \sqsupseteq C' \wedge C' \not\succ T$ should the loop invariants be updated; i.e., SPEC misses an incompleteness in the SPEC.

In fact, whether FAST is missing issues in the SPEC also depends on whether loop invariants, being more coupled with CODE, should be considered as SPEC or implementation details. As far as the authors know, there is no definite answer to this question and we are open to all views on this subject.

One data point we can offer is that loop invariants are indeed considered as SPEC by the DPN team as these invariants are not only hints for proving postconditions (as shown in Figure 9) but also contracts that need to be implemented in the loop body. In FAST, we actually had a mutation on line 44 from `i = i + 1` to `i = i + 2` and as expected, this causes failure in both the verification and testing. The rationale from the DPN team is that, should the loop be converted to a recursive function, loop invariants automatically become pre- and post-conditions (i.e., SPEC) for the converted function. Therefore, a failing loop invariant signals a robust SPEC.

An alternative approach in automated deductive verification to solve the complexity caused by loops is bounded model checking (BMC) which unroll loops to a certain depth, at the expense of completeness. A BMC-style verifier is compatible with FAST as the verifier can prove (or disprove) whether an arbitrary CODE mutant conforms to the SPEC.

```
1  fun add_members_internal<T: copy>(
2    members: &mut vector<T>,
3    to_add: &vector<T>,
4  ): bool {
5    let num_to_add = Vector::length(to_add);
6    let num_existing = Vector::length(members);
7
8    let i = 0;
9    while ({
10     spec {
11       invariant i <= num_to_add;
12
13       // the set can never reduce in size
14       invariant len(members) >= len(old(members));
15
16       // the current set maintains the uniqueness of the elements
17       invariant forall j in 0..len(members), k in 0..len(members):
18         members[j] == members[k] ==> j == k;
19
20       // the left-split of the current set is exactly the same as
21       // the original set
22       invariant forall j in 0..len(old(members)):
23         members[j] == old(members)[j];
24
25       // all elements in the the right-split of the current set is
26       // from the `to_add` vector
27       invariant forall j in len(old(members))..len(members):
28         contains(to_add[0..i], members[j]);
29
30       // the current set includes everything in `to_add` seen so far
31       invariant forall j in 0..i: contains(members, to_add[j]);
32
33       // having no new members means that all elements in the `to_add`
34       // vector seen so far are already in the existing set (vice versa)
35       invariant len(members) == len(old(members)) <==>
36         (forall j in 0..i: contains(old(members), to_add[j]));
37       };
38       (i < num_to_add)
39   }) {
40     let entry = Vector::borrow(to_add, i);
41     if (!Vector::contains(members, entry)) {
42       Vector::push_back(members, *entry);
43     };
44     i = i + 1;
45   };
46
47   Vector::length(members) > num_existing
48 }
49 spec add_members_internal {
50   // function never aborts
51   aborts_if false;
52
53   // everything in the `to_add` vector must be in the updated set
54   ensures forall e in to_add: contains(members, e);
55
56   // everything in the old set must remain in the updated set
57   ensures forall e in old(members): contains(members, e);
58
59   // everything in the updated set must come from either the old set
60   // or the `to_add` vector
61   ensures forall e in members:
62     (contains(old(members), e) || contains(to_add, e));
63
64   // returns whether a new element is added to the set
65   ensures result == (exists e in to_add: !contains(old(members), e));
66 }
```

Figure 9: A Move function with loop invariants