Finding False Assurance in Formal Verification of Software Systems

by

Ru Ji

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Abstract

Formal verification plays a crucial role in enhancing the reliability of computing systems by mathematically checking the correctness of a program. Although recent years have witnessed lots of research and applications that optimize the formal verification process, the issue of false assurance persists in certain stages of the formal verification pipeline. The false assurance problem is critical as it can easily undermine months if not years of verification efforts.

In this thesis, we first generalized the formal verification process. We then identified and analyzed specific stages susceptible to false assurance. Subsequently, a systematization of knowledge pertaining to the false assurance issues observed at these stages is provided, accompanied by a discussion on the existing defense mechanisms that are currently available.

Specifically, we focused on the problem of formal specification incompleteness. We presented FAST in this thesis, which is short for Fuzzing-Assisted Specification Testing. 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.

Acknowledgements

I would like to express my thanks to my supervisor Prof. Meng Xu, for all the academic support. I would also like to thank Prof. Yousra Aafer and Prof. Chengnian Sun for their time in reviewing my thesis and attending my presentation.

My sincere appreciation goes to the CrySP lab. There is no doubt that CrySP is by far the most lovely lab I have had the privilege to be a part of. To avoid inadvertently missing anyone, I choose not to list individual names. However, I am immensely grateful to each person who stood by me through challenging times, sharing joy and tears with me. I would not be able to make it this far without my labmates. The memories will forever hold a special place in my heart. I am also grateful to the professors in CrySP lab for immersing me in a great security research environment, and for being supportive and approachable to me when I needed help. I am happy to graduate from CrySP, having fulfilled not only my academic aspirations, but also gained significant personal growth.

Thanks to Jiaqi, Zhixin, Junjie, Ruiyi for being good friends over these years. They supported me through some desperate nights, and their daily companionship filled my life with happiness, their encouragement and belief in me held me up in my lowest days.

Also thanks to Lifei Wang for being a good friend. Thanks to Aries for being a lovely friend.

My deepest appreciation goes to my family. They always support all of my decisions and encourage me to explore more in my life. Their love becomes more invaluable when I am thousands of miles away from them. I want to give special thanks to my mom who gave me enough support to sort out whatever trouble I met in my life here in Canada, and she always had the patience for hours of late night phone calls with me that calmed me down.

Thanks to the CIF gym team, without which I cannot keep my workout habit.

The past two years have not been easy for me, but I still want to try my best to grow into a person who gives love to those around me.

Dedication

To my family.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Formal verification is a one-of-its-kind tool in enhancing the dependability of computing systems by mathematically proving their correctness — i.e., a system complies with its specifications under all possible conditions. Compared with other quality assurance tools such as testing, code review, and auditing, formal verification provides a superior dependability guarantee due to mathematical rigor, certainty of correctness, and a comprehensive coverage [79]. While still exotic in the general software development practice, in safety-critical industries such as avionics, formal verification is often either required or highly recommended by regulatory standards due to its thoroughness and rigor, making it indispensable for compliance [24].

As computing systems become increasingly integral to modern life, more software programs become safety-critical and the need for their dependability escalates, necessitating an increasing adoption of formal verification in recent years. In fact, formal verification has found applications across various sectors, both academically and industrially. Amazon Web Services (AWS), for instance, uses formal verification to bolster the security of its cloud infrastructure [40], showcasing a practical integration of formal verification in a commercial setting. The seL4 microkernel is another example, where formal verification underpins the implementation correctness of an operating system kernel without compromising performance [97].

Despite the guarantee of high assurance, formal verification is often perceived as a costly approach, requiring substantial human expertise and financial resources. To be specific, writing formal specifications often requires a completely different mindset of programming (declarative vs imperative) which usually necessitates additional training on the software developers. Specifications and proofs, if needed, are often developed in a lesser-known syntax and can sometimes be even more verbose than the implementation. In fact, even for the most qualified researchers, it is still a time-consuming and tedious task to formally verify any real-world program thoroughly. Illustrative of this is the example of seL4 [97]. A close examination of the verification effort statistics reveals the allocation of resources: 4 person-months for the abstract spec development, 2 person-years for the Haskell prototype, 2 person-months for implementing the Haskell prototype in C, and an astounding 20 person-years in the proof phase! The 20 person-years used in the proof phase includes 9 person-years invested in formal language frameworks, proof tools, proof automation, theorem prover extensions and libraries, and 11 person-years for seL4-specific proof. Not to mention that in the case where the codebase needs to be upgraded, based on the different types of changes, the cost also varies. In the worst case where new, large, cross-cutting features are added, chances are that it requires several person-years to re-verify. This underscores the detailed and demanding nature of formal verification both in time and expertise.

Frankly speaking, for most software programs, especially those with time-to-market pressure, adopting formal verification for dependability assurance can be prohibitively expensive. Nonetheless, in situations where software correctness is essential, developers are willing to take the high costs and resort to formal verification for the possibility of providing correctness guarantee to the largest extent. In the domain of smart contracts, the Certora Prover[3] offers a rule-based verification framework designed to identify vulnerabilities, logical inconsistencies, or departures from specified behaviors. Given the immutable nature of smart contracts upon deployment to the blockchain, any modification post-deployment is infeasible. Consequently, formal verification serves as a crucial preemptive measure to avert substantial financial losses attributable to flaws within smart contracts.

On the bright sight, integrating formal verification into the development cycle can emphasize the importance of correctness. The process of devising specifications and reasoning about the correctness of the program implementation compels developers to be more thorough and explicit on expressing their intention, which can enhance clarity and precision. This is important especially for subtle algorithms or cases with complicated state space. For example, in CPU design, formal verification can pinpoint subtle timing issues and rare conditions that traditional testing might miss. However, for non-technical stakeholders, the excessive cost, together with the complex and time-consuming process of implementing formal verification, may create an illusion that if a program is formally verified, the program's security is no longer a concern. **This assumption can be dangerously misleading!**

The truth is: even though formal verification is able to significantly reduce the chance of having uncovered bugs, it does not reduce the probability to zero. As perfectly illustrated by NASA in its report [2]:

We are very good at building complex software systems that work 95% of the time, but we do not know how to build complex software systems that are ultra-reliably safe (i.e. $P_f < 10^{-7}/hour$).

The quotation from the NASA report touches on a critical tension in the domain of formal verification: while formal verification offers a robust framework for ensuring that the software behaves as intended, it does not mean that the verified software is worry-free. This 5% gap is what we call **false assurance** in this thesis, and is one of the reasons why software that undergoes a rigorous process of formal verification can still be vulnerable.

False assurance can stem from various sources, including but not limited to incorrect or missing specifications, wrong assumptions about the environment, or flaws in the compilation or verification toolchains. In fact, in §3 we show that false assurance can arise from virtually every step in a formal verification process. On the other hand, the false assurance issue has been recognized and tackled by the community for decades with a rich set of tools and methodologies proposed to reduce or even mitigate false assurance hidden in different aspects of formal verification. On a high-level, these mechanisms can be categorized into two types: fuzzing (testing the procedure exhaustively) and modeling (adding another layer of verification to one of the procedures in the original verification process).

In this thesis, we first outline different flavors of formal verification and a generic workflow of the verification process (§2.1). We then present a catalog of false assurance that can possibly appear in the generic process. For each step prone to false assurance, we survey existing defense mechanisms as well as identify open problems (§3). Specifically, we focus on the problem of false assurance in formal specifications, and proposed a framework named FAST which can examine the incompleteness problem in an automated way (§4).

We would like to stress upfront that the goal of this thesis is NOT to undermine the utility of formal verification by cataloging potential points of failure. On the contrary, we intend to explore avenues for enhancing the level of assurance that formal verification can offer. Additionally, we aim to demystify the concept and foster broader adoption by elucidating its true significance. Crucially, our goal is to cultivate an informed understanding among stakeholders, particularly those not intimately involved with the verification and implementation processes, to ensure that the "formally verified" designation is neither misconstrued as an infallible guarantee nor overlooked for its substantial value.

Chapter 2

Background

2.1 Formal Verification

In this section, we give a brief introduction to formal verification for software systems, compare it with other software assurance approaches, describe common flavors of software verification, and present a conceptual workflow for the most common usage scenarios.

2.1.1 A Quest for Correctness

Correctness of programs has always been a major concern. Ensuring correctness is a fundamental aspect of software development. Correctness refers to the property of a software system that *operates according to its intended requirements under all defined conditions.*

For most software systems, the quest for correctness begins with a thorough understanding of the intended functionality and business requirements of the software, which forms the basis for all subsequent efforts:

① Rigorous and systematic testing [158], such as unit, integration, and system testing, ensures that the most critical aspects of the software perform as expected.

② Beyond basic testing, dynamic analysis, including fuzzing (for program state exploration) [113] and runtime verification (for bug oracle definition) [18], expand both state coverage and bug scope of execution-based approaches.

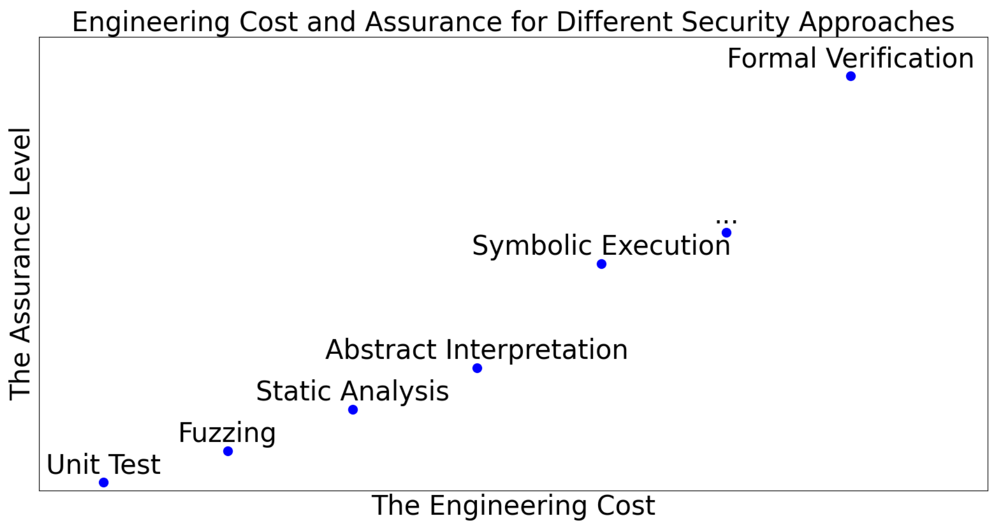

Figure 2.1: Trade-off between engineering cost and assurance level for security methods

③ Other than dynamic analysis, static analysis, such as declarative bug patterns (e.g., CodeQL [190]), abstract interpretation (e.g., Infer [4]), and symbolic execution (e.g., KLEE [31]), scale up the assurance guarantee from one concrete execution to an abstraction of program logic.

④ Ultimately, the application of formal methods provides a mathematical and logical approach to ascertain program correctness across all possible inputs and states with ideally no abstraction to program semantics.

While formal verification can provide the highest level of correctness assurance, in practice, it is rarely used, as the high assurance guarantee comes with extra engineering cost as demonstrated in Figure 2.1. Higher levels of assurance typically correlate with higher engineering costs. At the lower end of both cost and assurance is unit testing, which is characterized by minimal engineering effort but also provides the lowest level of assurance. Fuzz testing is noted for its feedback loop mechanism, implying a middle ground in terms of cost and assurance. Between the extremes of unit testing and formal verification, there exist various forms of static analysis, each likely offering a different balance of cost versus assurance. Hence, a practical consideration in choosing the right program correctness assurance method is balancing between engineering cost and level of assurance offered by various approaches, which also hinges on the specific needs of a project and the permissible engineering expenses.

2.1.2 Types of Formal Verification

By definition, formal verification is an approach used to check the correctness of a program *mathematically*. Based on the relationship between verification specification and the original program to be verified, formal verification framework can be categorized into model checking and deductive verification [100].

Model checking. Model checking takes the system to be analyzed as a state-transition system. The general approach of model checking is composed of three parts [38]:

❶ **Finite state-transition graphs** (S) which provide the formalism for the system.

❷ **Description of correctness properties** (φ) for state-transition systems.

❸ **Model-checking algorithm** which is a decision procedure for determining whether S conforms to the description φ.

In order to perform model checking on a system, the system has to be able to be modeled by state-transition systems. if $S \implies \varphi$, that means φ constrains S, otherwise if $S \not\Longrightarrow \varphi$, the model checker will output a counterexample in S which violates φ. It is easy to see from the description of model checking process that model checking face the problem of state explosion. Under most circumstances, there requires a bound to limit the number of states, which will make some of the bugs hidden in the states that have been pruned.

Deductive verification. Deductive verification approaches the challenge of ensuring that a program behaves correctly by requiring a more involved specification process. In contrast to model checking, which exhaustively explores the possible states of a system, deductive verification leverages logical reasoning to infer correctness. This method assumes a more comprehensive role for specifications, which necessitates a profound comprehension of the system under scrutiny to formulate accurate and detailed expectations of its behavior. The advantage of deductive verification lies in its capacity to deal with systems that have an infinite or very large state space, which model checking struggles with due to state explosion. This makes it particularly suitable for verifying properties of algorithms or systems where the number of states cannot be feasibly enumerated, as is often the case with complex software and hardware systems.

Deductive verification can be categorized based on the level of automation it incorporates. An elementary instance is the pen-and-paper proof [77], which refers to the most basic step-by-step reasoning on a program. This approach is used in the early attempt to verify programs. It is obvious that the deficiency with pen-and-paper approach is that it is inherently time-consuming and heavily reliant on the individuals conducting the proof. When it comes to validating the proof, it still requires extra human effort to validate the proof, thereby limiting its practical efficacy and scalability in the verification process. it also inherently carry a risk of incorrectness due to human error, rendering the guarantee of their correctness somewhat elusive. Furthermore, this traditional method lacks adaptability when dealing with code updates; changes in the code often require a comprehensive review and possible reconstruction of both the specification and the proof, these processes are generally largely manual and exhaustive. To overcome the limitations of the manual method, automated proofs have been developed. SMT solvers such as Z3 [43], CVC4 [16] are examples in this line of work as automatic theorem provers. However, automated proofs share the same deficiency as in model checking: if the search space for the given problem is too large, it will simply be too hard to validate. Besides, it is undecidable whether the automated provers can find proof for a given verification condition or not.

Between these two lines of work (pen-and-paper proof and automated proof), proof assistants, such as Isabelle [183], Coq [83], Agda [162] bridge human effort and computational power. Proof assistant works by requiring the programmer to input the major proof steps, and automatically provide assistance (validation of the reasoning steps, proved theorem storage, etc.,) when necessary.

2.1.3 Practical Concerns

There is no doubt that among security approaches, formal verification in general is a strict approach which can provide higher assurance. Nevertheless, this precision comes with its own set of challenges and costs. Scalability is one of the biggest challenges for formal verification. Ideally, every program should have complete mathematical proof. While the codebases nowadays are large enough that, big companies like Google are managing billions of lines of code, such that achieving such exhaustive verification becomes impractical. At the same time, proving a software system correct requires more effort and skills than writing the system itself. Consequently, formal verification is predominantly reserved where correctness is of crucial importance, and the codebase is not large enough that it will feasibly undergo the verification process (or sometimes, only part of the code will be formally verified). In scenarios where both software and hardware systems must ensure uncompromised correctness, formal verification emerges as the favored strategy to provide the utmost confidence in their reliability. Under such circumstances, formal verification is always the preferred solution to provide the highest possible guarantee of correctness.

2.1.4 Overall Workflow of Formal Verification

Despite the various different types of formal verification, the framework of formal verification can be described as shown in Figure 3.1, in summary, the overall workflow can be decomposed to the following steps:

Code transformation. Normally, source code will not be directly compared with the requirements. The conversion of program code into an abstract representation is an essential step in the verification process, especially in complex formal verification frameworks. This process allows the original code to be presented in a form that is more suitable for formal analysis. For example, in the framework of seL4 [97, 138], Haskell is chosen as the initial development language. The Haskell prototype is manually translated into C implementation as a compromise between high performance and convenience for formal verification developers. The use of Haskell is restricted to a subset that can be automatically translated into the language of the theorem prover. In this case, the translated C implementation enables optimization, and the Haskell prototype provides convenience for verification developers.

Requirement modeling. The requirement originated from the expectation of the behaviors in the program. It often requires a form of carrier to formally describe the generalized requirement into formalized description framework, in this way, the description can be fed into the formal verification framework. Generally, the requirement for the code is represented in some formal description language. There are different methods to model the requirement, based on different verification frameworks. These modeling approaches include pre-post condition modeling, state machine modeling, temporal logic modeling, etc. One example is predicate abstraction [1]. Predicate abstraction is an interpretation technique in which the abstract domain is constructed from a given set of predicates over program variables.

Build proof obligation. Formal specification and abstract code representation are used to construct proof obligation (a.k.a., verification condition). For example, Move [20] is a programming language developed for smart contracts by Meta. The language features formal verification at its core through its home-grown verification tool - Move Prover [53]. In a verification process, the input is the combination of move code and specification. The source code will be compiled into bytecode and the specification will be parsed into AST. These two parts will then go through a pipeline of merging and transformation, and finally be compiled into Boogie [14].

Verify proof obligation. Last but not least, the obligation is verified by verification tool. For the example used before, in Move framework, the verification conditions in Boogie format will be translated to SMT format which can be solved by SMT solver such as Z3 [43] or CVC4 [16]. As for model checking, popular model checkers include NuSMV, the first model checker to be based on Binary Decision Diagrams (BDDs) [37], Simple Promela Interpreter (SPIN) [81], the model checker especially used for concurrent and distributed systems. PRISM [103], a probabilistic model checker. Process Analysis Toolkit (PAT) [119, 171], a self-contained framework for composing, simulating and reasoning of concurrent, real-time systems and other possible domains. It includes user interfaces, a model editor and an animated simulator.

Chapter 3

False Assurance

Besides the scalability challenge, false assurance is a significant impediment that can easily undermine the efforts and investments in formal verification. For the part of the code that claims to be guarded by formal verification, false assurance refers to the failure to provide complete assurance; or the discrepancies between actual program implementation and formal specifications. Marton et al. [22] illustrated this through an inductive approach in two case studies. They collected falsifiable approaches at first, then tried to falsify the assumptions. The result uncovers gaps between formal, deductive-only approaches and practical security applications. The false assurance problem is notably prevalent in cryptography, where implementations are generally subjected to rigorous verification processes. Some works [46, 98, 99] summarized the problems of provable security as inadequate definitions, bodacious assumptions, and problematic proofs.

In this thesis, we want to focus more on the implementation side of false assurance than the overall mindset of programming formal verification. At the same time, we are providing a systematical analysis in the development of formal verification techniques and the according defense for the false assurance problem. Specifically, as presented in Figure 3.1, along the process of formal verification, false assurance can reside in several different points, the false assurance will also propagate along different layers of the process. We generalize and pinpoint the following false assurance locations:

❶ **Specification**. The program specifications are a set of expectations for the behavior of the program. In the context of formal verification, the specifications provide the expectations and instructions for the verification framework to check the implementation. However, there's a risk that specifications might be incomplete or inaccurately reflect the intended requirements, which can lead to false assurance.

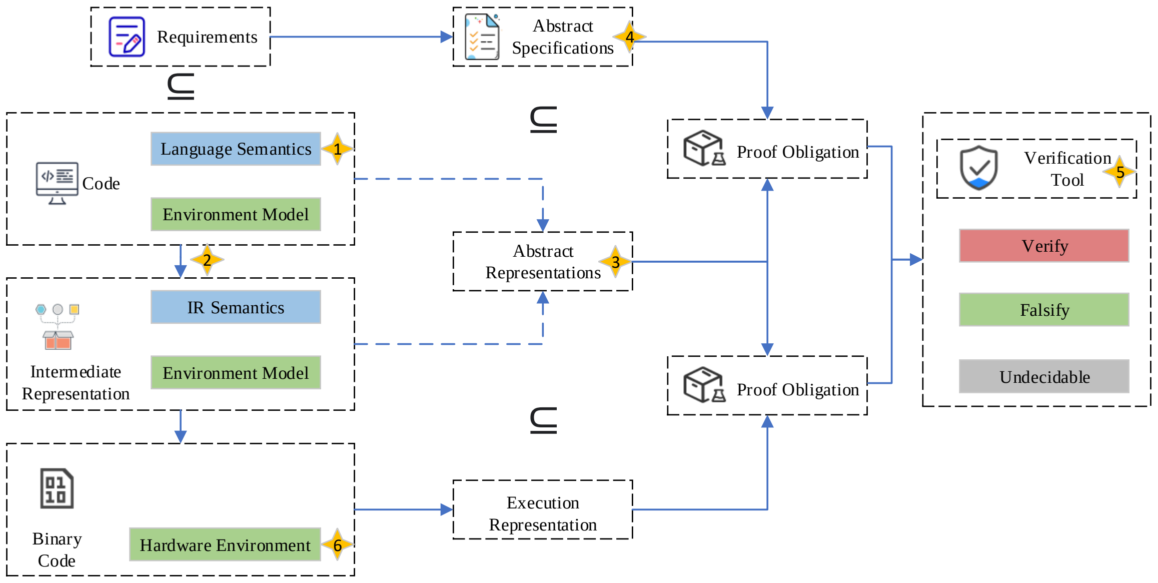

Figure 3.1: Formal verification workflow with false assurance

❷ **Code semantics**. Programmers can have different understandings of the semantics of code. For most projects, the standard of interpreting the semantics is from the official document. However, different programmers can interpret it in different ways, or there might be mistakes in the document itself. Such disparities can cause a misalignment in the formal specification process, contributing to false assurance.

❸ **Abstract representation**. Abstractions in code are crucial for scalability, maintenance and readability. For example, in the translation validation process in seL4, the C source code is parsed into higher order logic representation (HOL), and subsequently converted into graph language. This graph is then matched against a version derived from the binary code to ensure the correct compilation. However, the tools used for these conversions are not infallible and may introduce inaccuracies.

❹ **Code transformation**. Usually, source code will not be directly used as an input for verification, source code typically undergoes transformation to align with the formal verification model. In this process, the transformation tool can be problematic that it will lead to discrepancies between the transformed code and the target code.

❺ **Verification tool**. The proof obligation that comes from all the preprocessing steps will eventually be handled by the verification tool, which determines their validity. For example, theorem prover in deductive proofs is a form of verification tools.

❻ **Hardware**. The source code gets compiled into binary code which will be executed on hardware. The reliability of the hardware itself is pivotal; any underlying bugs can compromise the correctness of the program execution, thus leading to false assurance.

Based on the pipeline described in Figure 3.1, here we elaborate more on each false assurance point, with examples and the analysis of current research on the discovery of false assurance and potential defense mechanisms. The defense mechanisms have been generalized into Table 3.2. The criteria for paper selection is listed in the appendix. In the following table, the number of bugs found labeled with * is not direct bug discovery, but potential optimization on performance; The number of bugs labeled with [-c] is the number of errors reported from the tool provided in the papers, but has not been officially confirmed. e.g., it can be the number of failed generated tests by the testing subject.

3.1 Specification

Developing a program starts with conceptual requirements. However, there is no guarantee that the basic requirements or assumptions for the program to be verified will be established correctly.

In formal verification, the conversion from conceptual requirements into formal specifications is critical, yet inherently fraught with challenges. There is no absolute assurance that the original requirements are flawless or that the derived specifications comprehensively encapsulate the intended program functionality. This potential for incorrectness and incompleteness during the transition from conceptual requirements to formal specifications can introduce risks, particularly in the verification of expansive systems. Such systems often necessitate underlying assumptions or the omission of certain code segments, further compounding the possibility of false assurance. The fidelity of translating conceptual requirements to formal specifications is thus not infallible, and discrepancies in conveying the true intent are a concern. Here we take an example of functional verification to illustrate the specification incompleteness problem in formal verification. In this example shown in Figure 3.2a, the developer intends to increment each element in the vector by one. The specification ensures that each element in the return vector will get incremented by one compared with the vector in the precondition. However, the specification shown in Figure 3.2b fails to also monitor that the lengths of the vector from precondition and post-condition have to be the same. Under the current setting, should the function erroneously eliminate an element, the verification could erroneously succeed due to the oversight in the specification. This exemplifies how a seemingly comprehensive verification process can

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

(a) CODE for `add`

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

(b) SPEC for `add`

Figure 3.2: Demonstration of specification incompleteness

overlook critical aspects, resulting in a misalignment between the verified behavior and the intended functionality of the program.

3.1.1 Gauging Incompleteness

The idea of finding discrepancies between specifications and the actual implementation is known as gauging specification incompleteness in the literature [159, 173]. Some research works try to measure the quality of specifications by heuristic based approaches. Leveson et al. identified a set of formal criteria to identify missing, incorrect, and ambiguous requirements for process-control systems [85, 112, 137]. Csertan et al. introduced a reduced form of state charts in order to automate the verification process [41]. Most recently, Bognar et al [22] use 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. Specifically, they first identify falsifiable assumptions about the system behavior, then validate whether the assumption holds in the real system and if not, try to exploit missing attacker capabilities.

3.1.2 Mutation Testing

The heuristic based approach requires lots of manual effort. In the hardware verification context, mutation testing [147] is used to solve the same problem in a more automated way. Because formal verification has a longer history in hardware design compared with software design, mutation testing has been applied to improve the completeness of hardware specifications, we see implementations on digital circuits [102, 177], processor [66], etc.

However, in the software verification context, the implementation of formal verification is less common. There has been earlier research work on specification mutation [89]. We can see the applications for mutation testing on specification and description language(SDL) [167], model language [51], etc. FAST [88] proposed an evolutionary mutant generation approach, it also leverages the test suites to infer whether the gap is introduced by intention or by mistake. FAST [88] works by firstly locating SPEC gaps via mutation testing, i.e., by checking whether a CODE variant conforms to the original SPEC. If CODE conforms to the SPEC, FAST will leverage 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 way or an evolutionary way.

For model checkers, mutation testing is also used to check the incompleteness of properties that are being checked in the model checkers. Example includes NuSMV [71], some FSMs [11, 151].

3.2 Semantics

The official reference for most programming languages is the manual. However, although manuals can provide a formal syntax definition, there is no formal semantics definition, which means that the semantics of the source code can be interpreted in different ways. This can bring extra trial and error for the developers and will lead to possible false assurance.

```cpp
std::size_t func(int x)
{
    std::size_t a;
    if (x)
        a = 5;
    return a;
}
```

In the above example, this C++ code snippet represents a function where the return value will be given based on the value of the one parameter that is taken into the function. If the parameter is non-zero, the return value will be 5, otherwise it will be an undefined behavior based on the definition of C++. In this case, there is a chance of getting undefined behavior if variable x is provided with the value 0 and the if branch is not reached. Generally, programmers are expected to prevent undefined behavior from happening. Compilers are not required to diagnose undefined behavior, and the compiled program is not required to do anything meaningful. However, when undefined behavior like this is passed on to the next layer, it can lead to errors in the next several layers.

3.2.1 Formal Semantics

One way to mitigate the semantic problem is by providing a formal framework for programming language semantics. One example on this line of work is K-framework. K-framework

[156, 157, 164] is a tool-supported rewrite-based framework for defining programming language design and semantics. K offers several implementation scenarios including interactive execution (kcompile, kparse), state space exploration (krun), and deductive program verification(kprove), etc. K-framework has also led to other works, K-LLVM [114] provides complete formal LLVM IR semantics [192].

3.2.2 Fuzzing

Zest [146] adapts the algorithm used by CFG tools in order to quickly explore the semantic analysis stages of test programs. Fuzzing has also been used on specific applications targeting semantic bugs, for example, TCP stacks [194], file systems [96], web browser [193].

3.3 Transformation

As shown in Figure 3.1, source code normally will not be directly used in generating proof obligations, it will go through a transformation process. Compilation is one of the transformation approaches. However, the compiler might provide false assurance since it can not be guaranteed to be bug free. The consequence is that compiler can result in compiled code that has discrepancies with the source code. Analysis of compiler bugs in GCC and LLVM [170] shows that GCC has 39,890 reported bugs in a time range of 16 years. LLVM has 12,842 in a time range of 12 years. C compiler has a longer history of being a research object. In order to better illustrate the potential optimization problem, here is an optimization error example in LLVM. Given a code piece:

```
a + b > a
```

The LLVM IR code before any optimization is as below:

```
%add = add nsw %a, %b
%cmp = icmp sgt %add, %a
```

However, after the optimization, the optimized code is:

```
%cmp = icmp sgt %b, 0
```

the equivalent code will be:

```
b > 0
```

In this case, when `a` is assigned with the value `INT_MAX`, and `b` is assigned with `1`, the result of `a + b` should lead to value overflow, which is an undefined behavior[109] in LLVM. However, since it is optimized to `b > 0`, the value of `a` does not matter anymore. This line of code will always be evaluated to be valid. Therefore the optimization adds incorrectness into the process. The correctness of code transformation process is a concern, it is a hidden threat to program safety.

3.3.1 Formal Verification

One of the idealistic ideas is to secure the compiler with formal verification. There have been mature projects targeting formally verifying the compiler implementations of different programming languages with the hope to be exempted from miscompilation. For C, CompCert [17, 111, 174] is an optimized compiler where the executable code produced is proved to behave exactly as specified by the semantics of the source C program, Frama-C [42] is a source code analysis platform that aims at conducting verification of industrial-size C programs. ESC/Java [69] is an experimental compile-time program checker that finds common programming errors. It is powered by verification-condition generation and automatic theorem-proving techniques. Spec# [15] is a programming system which is consisted of Spec# programming language, Spec# compiler and the Boogie static program verifier. The compiler can emit run-time checks to enforce the specifications, and the verifier can check the consistency between the program and it's specifications. CertiCoq [5] is a mechanically verified, optimized compiler for Coq that bridges the gap between certified high-level programs and their translation to machine language. Rustbelt[90] is designed for formally verifying Rust compiler. Some work targets on creating a formalized model on some specific issues. VCC [39] focuses on verifying c programs in concurrent scenarios, formalizing the racy access semantics of an LLVM fragment [33]. Vellvm [192] (verified LLVM), is a framework for reasoning about programs expressed in LLVM's intermediate representation and transformations that operate on it. The framework is built using the Coq interactive theorem prover, and provides a mechanized formal semantics of LLVM's intermediate representation, its type system, and properties of its SSA form. Crellvm [91] augment an LLVM optimizer to generate translation results together with their correctness proofs, which can then be checked by a proof checker formally verified in Coq.

3.3.2 Translation Validation

Translation Validation [165] looks at the problem of miscompilation from another perspective. Formal verification tries to prove in advance that the compiler always produce target code that can correctly convey the functionality of source code. While translation validation can provide more usability by hiding the tedious formal verification workload. The Alive line of work focuses on handling undefined behavior in LLVM optimized IR. Lee et al. summarized the undefined behavior design in LLVM [109]. Alive [122] is a domain-specific language for writing optimizations and prove the correctness of the optimizations. Furthermore, Alive translates into C++ code that implements the optimization pass. AliveInLean [107] took one step further from Alive in that it specified and verified Alive in Lean [9]. Alive2 [121]can take an unoptimized function in LLVM IR and either prove that it is refined by the optimized version of the function or show a set of inputs that illustrate a failure of refinement, taking undefined behavior into consideration. Alive2 also proposes its own memory model [108]. The similar approach has also been used on other compilers, including machine learning compilers [12]. CORK [123] is an automatic equivalence checker that supports loop optimizations over rational numbers.

3.3.3 Fuzzing

It is obvious that formal verification is going to take lots of effort when trying to verify a compiler, especially with the constant upgrade and possibly new variations. Another more intuitive way to test the correctness of compiler is to generate test cases and feed the compiler with the generated test cases. randprog is such a generator [176]. However, this generator itself has many limitations. In a large code base as compiler, we will want to be able to reach a test case that is closer to vulnerability in the code base. In several following works, randprog is used as a base tool, with optimizations on the deficiencies. Eric and John extended randprog by enabling the availability of finding volatile bugs with access summary testing [58] with the tool named volcheck. Except for testing the correctness of programs themselves, there are works that focus on a certain functionality of the program. For example, targeting testing the correctness compiler's local optimization rules, Optgen [26] can automatically generate peephole optimizations. CSmith [189] compared with randprog has lots of optimizations, including complex control flow and supports for data structures such as pointers, arrays, structs, etc. YARPGen [120] took one step further from CSmith by including a method for generating expressive programs that avoid undefined behavior without using dynamic checks, and implementing generation policies to increase the diversity of generated code, thus triggering more optimizations. Besides research based on randprog, there are other testing approaches with more limited test generation constraints. Le et al. proposed equivalence modulo inputs(EMI) [104], which tries to exploit the interplay between dynamically executing a program P on a subset of inputs and statically compiling P to work on all inputs. To be specific, EMI works by modifying unexecuted parts of the seed program with the information provided by code coverage tools. This simple strategy of deleting statements, which is more succinct in terms of code size compared with Csmith work well in practice. This approach is novel in that it is easily applicable, and it can generate test cases based on real-world code. The EMI test approach has been later extended in several works. Both Orion and Athena perform EMI by deleting code from or inserting code into regions that are not executed under the inputs. Athena [105] based on Orion, with enabling code insertion into unexecuted program regions and using Markov Chain Monte Carlo(MCMC) techniques to guide the generation process. Proteus [106] utilizes the techniques from both CSmith and Orion to test Link-time optimization (LTO). Epiphron [168] targets compiler diagnostic warnings. It leverages the techniques from Orion and solved several challenges that are specifically for compilers' warning, including aligning warnings, reducing test cases, and generating effective test programs. Hermes [169] mitigates the limitation by allowing mutation in the entire program.

3.4 Verification Tools

The proof obligation has to go through verification tools in order to get the final result. Depending on the specific type of formal verification, the verification approach also varies. But it is an essential step in order to validate the proof obligation. In terms of deductive verification, the correctness of SAT, SMT provers has gained lots of focus. As for model checking approaches, the correctness and performance of model checkers are the focused points.

When it comes to the evaluation of formal verification tools, soundness, completeness, performance are the usual aspects to be judged upon.

3.4.1 Fuzzing

FuzzSMT [25] is a blackbox grammar-based fuzzer for generating syntactically valid SMT instances. FuzzSMT is unable to preserve satisfiability at the same time. StringFuzz [21] fuzzes SMT solver with string and regex constraints. BanditFuzz [163] took one step further than StringFuzz, using reinforcement learning to provide better performance than

StringFuzz. Bugariu et al [30] targets the problem of not able to automatically classify the result as true or false by synthesizing input formulas that are satisfiable or unsatisfiable by construction and use it as ground truth in test oracle. Dominik et al. proposed a semantic fusion approach. Semantic fusion is a general and effective way to solve the problem of generating test formulas to validate SMT solvers [186]. OpFuzz [184] uses type-aware operator mutation by mutating the operator within the seed formulas to generate well-typed mutant formulas. STORM [128] serves as a blackbox mutational fuzzing technique, which takes inspiration from AFL and generates new SMT instances by mutating existing ones. TypeFuzz [148] combines mutational and grammar-based type-aware fuzzing. To be specific, given a seed formula ϕ, they firstly choose an expression within ϕ, then pick an operator of the same type as the expression and fill operator's arguments with expressions from ϕ.

3.4.2 Formal Verification

As for model checking approaches, it is more common to see that model checkers have been formally verified to add more assurance. Esparza et al [62] presented an LTL model checker that has been completely verified using the Isabelle theorem prover. Formal verification has also been used in verifying interactive proof assistance tools. SMTCoq [61, 92] increases the level of automation of Coq by automatically calling external solvers and checking their answers to solve a class of Coq goals; while at the same time provides an independent and certified checker for SAT and SMT proof witnesses. It was further extended to support more use cases and different SMT solvers, or provide new tactics to the existing solvers [13, 60]. Sozeau et al [166] presented a type checker for the kernel of Coq, which is proven correct in Coq with respect to its formal specification and axiomatisation of part of its metatheory.

3.5 Hardware

False assurance in hardware lies in the fact that many developers think once the program is handed to hardware, there will be no problem afterward [93]. However, real-world cases show that the hardware assurance is still problematic. SHADE [87] is a tool that targets false assurance from automated visual inspection approach. In this paper they proposed an algorithm which exploits shadows cast by surface-mount components to accurately distinguish them from their invalid counterparts (e.g. traces, vias, board text) Since the consequence of having errors in hardware is more intuitive, hardware has long been a target of formal verifications. It is a wild-range concept therefore the target for formal verification

#	Tool	Confirmed Bugs	Target Type	Target	Category	No.
1	Bognar et al [22]	9+8	Specification	Embedded architecture	Heuristic	4
2	FAST [88]	13+21	Specification	Move, s2n-tools	Fuzzing	4
3	K-framework [156]	-	Language Semantics	Programming language	Modeling	1
4	CompCert [111]	-	Compiler	C	Formal Verification	2
5	Frama-C [19]	-	Compiler	C	Formal Verification	2
6	Spec# [15]	-	Compiler	C#	Formal Verification	2
7	ESC/Java [69]	-	Compiler	Java	Formal Verification	2
8	CertiCoq [5]	-	Compiler	Coq	Formal Verification	2
9	VCC [39]	-	Compiler	C in concurrency	Formal Verification	2
10	RustBelt [90]	-	Compiler	Rust	Formal Verification	2
11	Vellvm [192]	-	Compiler	LLVM	Formal Verification	2
12	Crellvm [91]	-	Compiler	LLVM	Formal Verification	2
13	DIWI [34]	90	Compiler	GCC; LLVM	Mutation Testing	2
14	Alive [122]	8	Compiler	LLVM	Translation Validation	2
15	AliveInLean [107]	-	Compiler	LLVM	Formal Verification	2
16	StringFuzz [21]	0*	SMT Solver	Norn; CVC4; Z3	Fuzzing	5
17	BanditFuzz [163]	0*	SMT Solver	Z3;CVC4; Bitwuzla	Fuzzing	5
18	Bugariu et al [30]	1364 + 773 + 338 + 949[-c]*	SMT Solver	Z3-seq; Z3str3; CVC4;MT-ABC	Program Synthesis	5
19	STORM [128]	29	SMT Solver	Boolector, CVC4, Math-SAT5, SMTInterpol, STP, Yices2, Z3	Fuzzing	5

Table 3.1: Summarization of defense mechanisms

#	Tool	Confirmed Bugs	Target Type	Target	Category	No.
20	OpFuzz [184]	819	SMT Solver	Z3; CVC4	Mutation testing	5
21	YARPGen [120]	83 + 62 + 76	Compiler	GCC;LLVM;Intel Compiler	Fuzzing	2
22	volcheck [58]	[-c]	Compiler	GCC;LLVM;Intel Compiler	Fuzzing	2
23	Csmith [189]	325	Compiler	GCC; LLVM; CIL; TCC; Open64; CompCert; Commercial products	Test Generation	2
24	EMI [104]	147	Compiler	GCC; LLVM	Mutation Testing	2
25	Orion [104]	79 + 68	Compiler	GCC; LLVM	Mutation Testing	2
26	Proteus [106]	37	Link-time optimization(LTO)	GCC; LLVM	Fuzzing	2
27	classfuzz [35]	62	Virtual Machine	JVM	Fuzzing	2
28	Athena [105]	72	Compiler	GCC; LLVM	Fuzzing	2
29	Epiphron [168]	60	Compiler	GCC; Clang	Fuzzing	2
30	Hermes [169]	168	Compiler	GCC; LLVM	Fuzzing	2
31	Yinyang [186]	45	SMT Solver	Z3, CVC4	Fuzzing	5
32	TypeFuzz [148]	189	SMT Solver	Z3 CVC4	Fuzzing	5
33	Esparza et al [62]	-	Model Checker	Model Checker implemented in ML	Formal Verification	5
34	SMTCoq [61]	-	Proof Assistant	Coq	Formal Verification	5
35	Sozeau et al [166]	-	Proof Assistant	Coq	Formal Verification	5

Table 3.2: Summarization of defense mechanisms (continued)

comes in different perspectives. The recent popular discussion about Rowhammer[139], which is a hardware security vulnerability that affects dynamic random-access memory (DRAM) chips, stems from the miniaturization of DRAM cells, which has made them more vulnerable to electromagnetic interference from neighboring cells. Rowhammer provides a demonstration of the possibility that a circuit-level failure mechanism can cause a system security vulnerability with crucial consequences. The mitigations for Rowhammer include adding detection in the memory controller or refreshing the potential victim to lower the chance of being targeted. PROTRR provides in-DRAM Target Row Refresh mitigation with formal security guarantees and high performance [118, 129, 130, 150].

3.5.1 Formal Verification

For the hardware description languages, the common languages usually have extensive work on formal verification. Hardware description languages includes Verilog [6, 144], VHDL [95], SystemC [117]. Some tools also provide formal verification functionality during the development procedure[154], including JasperGold, OneSpin, etc.

3.5.2 Fuzzing

Exhaustive approaches are also used in detecting bugs in hardware description languages [59, 116]. Register transfer level (RTL) hardware is not inherently executable. Therefore, they can be simulated in software model, where the problems in hardware fuzzing can be solved via the approaches used in software fuzzing [175]. For example, the fuzzer is able to provide coverage metrics in the software model; the software model is also able to provide a hardware equivalent crash.

Chapter 4

Finding Specification Blind Spots via Fuzz Testing

4.1 Introduction

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 thesis, 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 [44, 70, 135, 148, 185, 186], 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 [65, 66], 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 thesis (§4.6.1 and §4.6.2), 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 [64, 172], 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 systems. 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?

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.

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 [141]), 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 thesis, 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.

4.2 Background

In this section, we give a brief introduction to 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 form them.

4.2.1 Automated Function Verification

While FAST can be applied to different flavors of formal verification (e.g., protocol verification [80], state-machine transitions [54, 134], etc.), in this thesis, we focus on a specific type of verification: functional correctness verification with preconditions and postconditions, sometimes also known as “design-by-contract” [132].

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 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 [76], Kani [8] VeriFast [84]), or *interactive verification* in which both the SPEC and a majority of the proof needs to be developed manually (e.g., HOL [74], Isabelle [183], Coq [83]). 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 3.2 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

```
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 4.1: Finding the gap in SPEC via CODE mutant

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.

4.2.2 The Completeness of Specifications

While it is obvious that in Figure 3.2 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 4.1a, 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 4.1b 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 3.2 and 4.1 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 [140]. 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 *gauging the completeness of* SPEC in the literature [159, 173], 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 [22]. In the hardware verification context, mutation testing is a more popular way to gauge the incompleteness gap, as will be described in §4.2.3,

4.2.3 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 [187]. Since its inception in 1970s [27, 49, 78], 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 [178], integration testing [47, 75], end-to-end testing [147], 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++ [48], Java [125], Ruby [115]. Mutation testing can be used to ensure the effectiveness of large applications developed with multiple languages as well, such as web applications [153] and Android applications [136]. 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) [63, 126] 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 [66, 102, 177]. 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 [10, 89, 133, 167] considered CODE as a black-box, and mutate the SPEC in order to find out incompleteness in SPEC [28], There are some implementations on FSM-based SPEC [45, 152].

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 [29], applying selective mutations [131], or adopting heuristics and search-based [188] 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 §5.

4.2.4 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 [127]. 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 [191] 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 [23, 68, 124, 155, 161, 179].

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 [127]. 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 [182] is an AFL-based fuzzer to find the bugs in XML and JavaScript engines. There are also other works aiming at JavaScript engines [55, 149], and Java language [86, 94]. 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 [52, 57], some other works use different feedback patterns (e.g. code coverage) to guide the fuzzing mutant[82, 145].

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.

4.3 The Tale of SPEC, CODE, and Tests

As demonstrated in the language fuzzing work (§4.2.4), 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 4.1a), **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 [101, 102], 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 3.2 and 4.1, C, S, and C' are shown in Figure 3.2a, 3.2b, and 4.1a, 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 thesis, we focus on mutation testing to create Δ_C. More discussion on SPEC mutation can be found in §5.

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 3.2, 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.

We provide more background information on the two case studies covered in §4.6.1 and §4.6.2 respectively as well as a discussion on how loop invariants might have an impact on FAST in automatically confirming gaps in SPEC.

4.4 Background of the Case Studies

4.4.1 Background on Diem Payment System

The Move programming language. Move [20] is a programming language developed for smart contracts by Meta although it has transitioned into a community-backed project now.

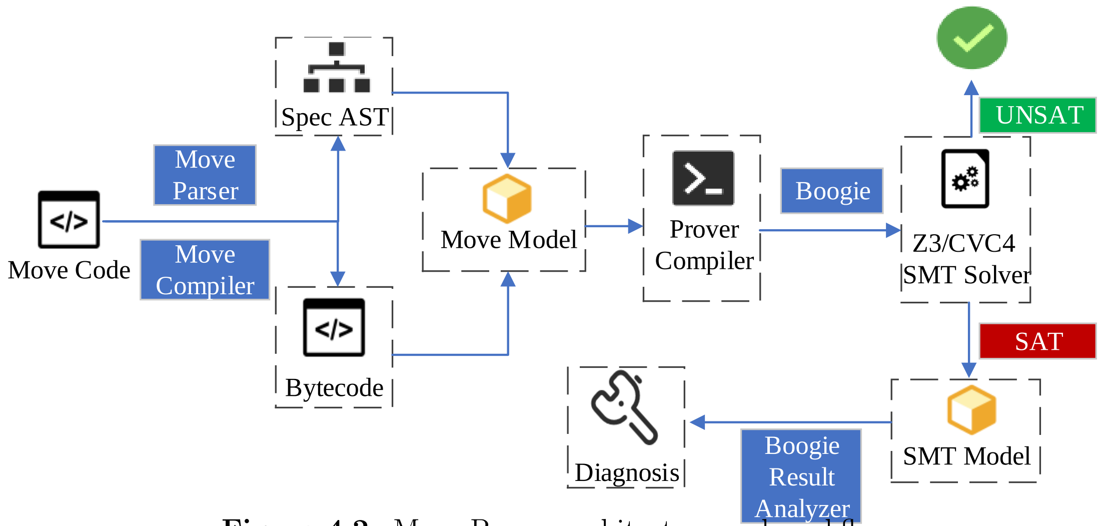

Figure 4.2: Move Prover architecture and workflow

The language features formal verification at its core through its home-grown verification tool Move Prover [53], which statically verifies the correctness of Move smart contracts modeled with the Move Specification Language.

The Move Prover. The architecture of the Move Prover is shown in Figure 4.2. 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 [14], 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 [44] or CVC4 [50].

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%.

4.4.2 Background on AWS TLS Implementation

About S2N. Amazon S2N-TLS [160] 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 [56]. 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 4.3. 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 [72] behind the scenes.

4.4.3 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 4.4.

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 $S \not\sqsupseteq C' \land C' \not\succ T$ (case ⑤ in §4.3), it might actually be case ④ $S \sqsupseteq C' \land C' \not\succ T$ 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.

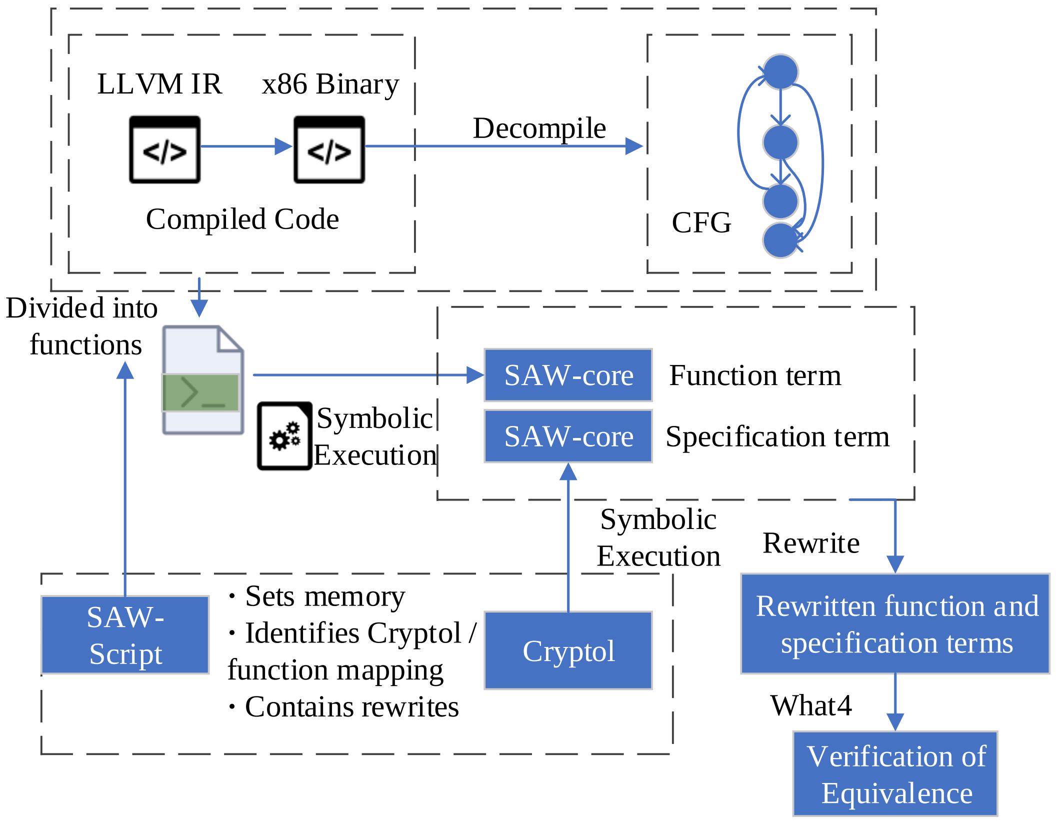

Figure 4.3: SAW architecture and workflow

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 4.4) 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.

4.5 Design of Case Studies

4.5.1 Enumerative CODE Mutant Generation

With the SPEC gap classification problem solved in §4.3, 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 §4.5.2. 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 4.1 shows 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 §4.6.1.

Enumerative algorithm. Given the limited set of mutation rules discussed in Table 4.1, for small codebases that do not 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.

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

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
```

CODE mutant by mutating more than one instruction 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 §4.7). FAST found mixed evidence on coupling effect in our case studies.

4.5.2 Evolutionary CODE Mutant Generation

As shown in §4.6.1, 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 grows 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 4.1. Similar to the enumerative approach (§4.5.1), 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 4.5 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.

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.

4.6 Implementation of Case Studies

4.6.1 Case Study: Diem Payment Network

Being a small yet critical smart contract, the Diem Payment Network [7] 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 4.6 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 4.1 to produce CODE mutant. For constant mutations, CODE randomly picks one of the three mutation rules listed in Table 4.1 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 4.2, 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.
- 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 4.2). 8 mutants passed the tests unconditionally, out of which 2 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 in case 2-4).

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 $S \sqsupseteq C' \wedge C' \not\vdash T$, 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 4.1 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 $S \sqsupseteq C' \wedge C' \succ T \wedge T \not\leq \Delta_C$. As a result, although both CODE mutants pass verification, they are considered to be an intended gap in SPEC.

4.6.2 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 4.5 shows the overall fuzzing process implemented in FAST, Figure 4.7 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 4.5 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 the 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 in case 4). These findings are summarized in Table 4.3. We have reported all findings to the development team of S2N. The team has acknowledged all reports and we are currently waiting for their patches. (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.

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;
```

```c
}

// 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:

```c
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:

```c
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
```

4.7 Extra Evaluations

While the effectiveness and practicality of FAST is evaluated on the two real-world case studies (§4.6.1 and §4.6.2), 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 4.2 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 4.3 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 [49]. 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 [142] and theoretical [180, 181] 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 4.8 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 rate of producing new surviving CODE mutants decreases gradually over time until reaching a saturation point.

```
fun add_members_internal<T: copy>(
  members: &mut vector<T>,
  to_add: &vector<T>,
): bool {
  let num_to_add = Vector::length(to_add);
  let num_existing = Vector::length(members);

  let i = 0;
  while ({
    spec {
      invariant i <= num_to_add;

      // the set can never reduce in size
      invariant len(members) >= len(old(members));

      // the current set maintains the uniqueness of the elements
      invariant forall j in 0..len(members), k in 0..len(members):
        members[j] == members[k] ==> j == k;

      // the left-split of the current set is exactly the same as
      // the original set
      invariant forall j in 0..len(old(members)):
        members[j] == old(members)[j];

      // all elements in the the right-split of the current set is
      // from the `to_add` vector
      invariant forall j in len(old(members))..len(members):
        contains(to_add[0..i], members[j]);

      // the current set includes everything in `to_add` seen so far
      invariant forall j in 0..i: contains(members, to_add[j]);

      // having no new members means that all elements in the `to_add`
      // vector seen so far are already in the existing set (vice versa)
      invariant len(members) == len(old(members)) <==>
        (forall j in 0..i: contains(old(members), to_add[j]));
    };
    (i < num_to_add)
  }) {
    let entry = Vector::borrow(to_add, i);
    if (!Vector::contains(members, entry)) {
      Vector::push_back(members, *entry);
    };
    i = i + 1;
  };

  Vector::length(members) > num_existing
}
spec add_members_internal {
  // function never aborts
  aborts_if false;

  // everything in the `to_add` vector must be in the updated set
  ensures forall e in to_add: contains(members, e);

  // everything in the old set must remain in the updated set
  ensures forall e in old(members): contains(members, e);

  // everything in the updated set must come from either the old set
  // or the `to_add` vector
  ensures forall e in members:
    (contains(old(members), e) || contains(to_add, e));

  // returns whether a new element is added to the set
  ensures result == (exists e in to_add: !contains(old(members), e));
}
```

Figure 4.4: A Move function with loop invariants

#	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 `|`, ˆ
9		BitOr	`|`	One of `&`, ˆ
10		BitXor	ˆ	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 4.1: Generic CODE mutation rules available in FAST

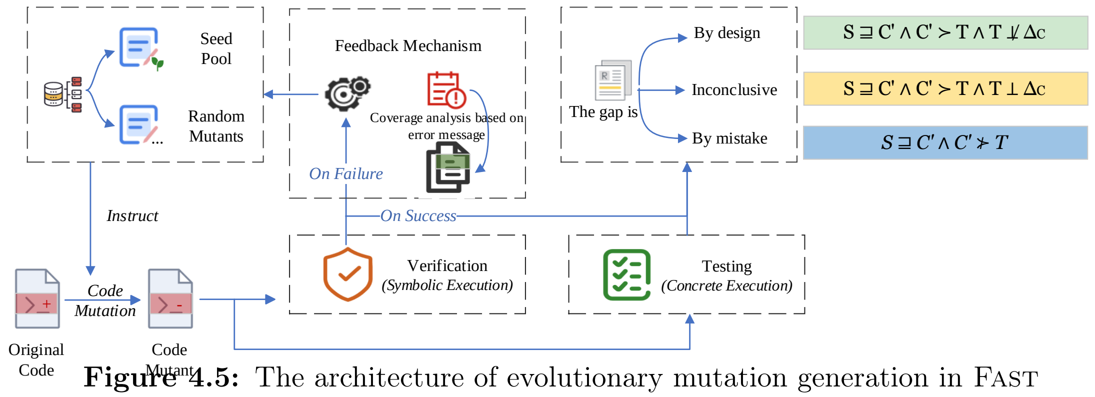

Figure 4.5: The architecture of evolutionary mutation generation in FAST

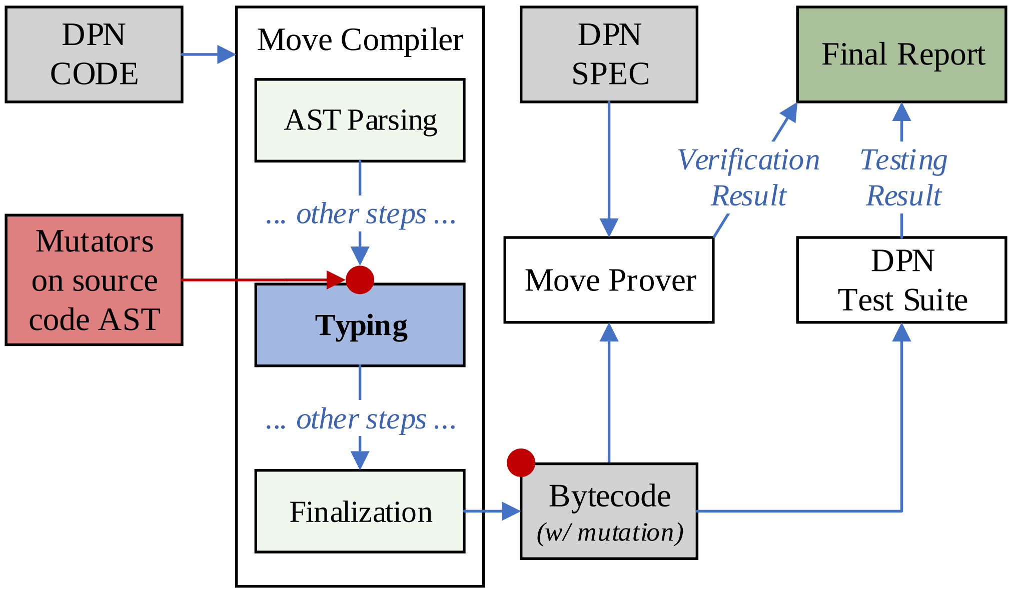

Figure 4.6: CODE mutation pass in the Move Prover pipeline

#	File Name	Function Name	Mutation Point	Mutation Rule	Test	$\mathbb{C}$	§4.3	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 4.2: Findings on the DPN case study. Issues 1-13 are reported and fixed while 14-16 are confirmed to have no harm.

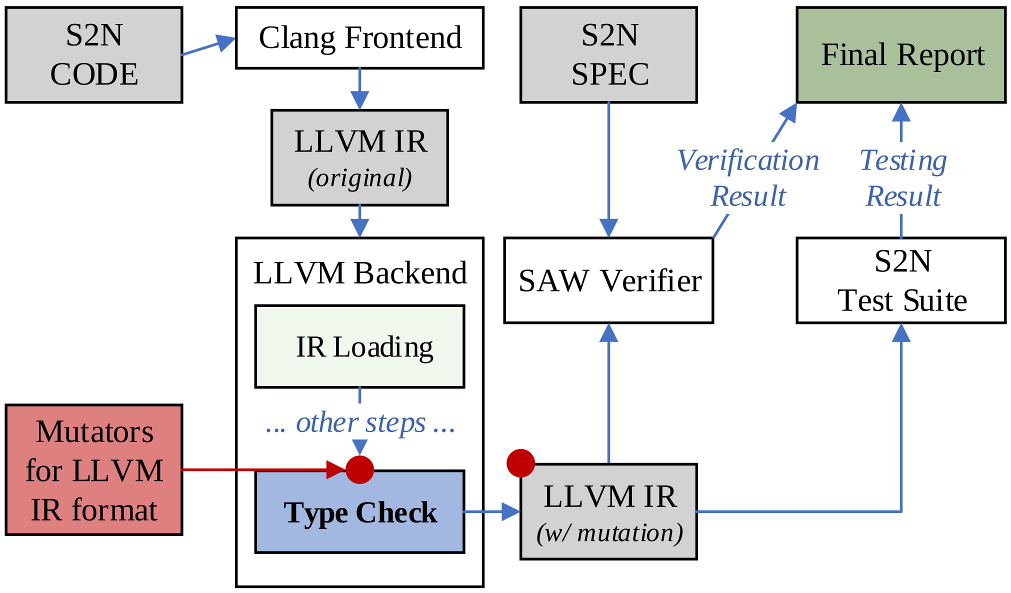

Figure 4.7: S2N mutation architecture

#	File Name	Function Name	Mutation Point	Mutation Rule	Test	ℂ	§4.3	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_increment_drbg_counter`	Add	Shl	Fail	✓	④	
	s2n_blob.c	`s2n_blob_validate`	BitAnd	LShr				
21	s2n_fork_detection.c	`s2n_get_fork_generation_number`	Constant	/= 3	Fail	✓	④	
	s2n_socket.c	`s2n_socket_was_corked`	BitAnd	Add				
22	s2n_handshake_io.c	`s2n_conn_set_tls13_handshake_type`	Constant	+=1	Pass	✓	①	Case 4

Table 4.3: The sampled results from the S2N case study. All reports have been submitted to the S2N development team.

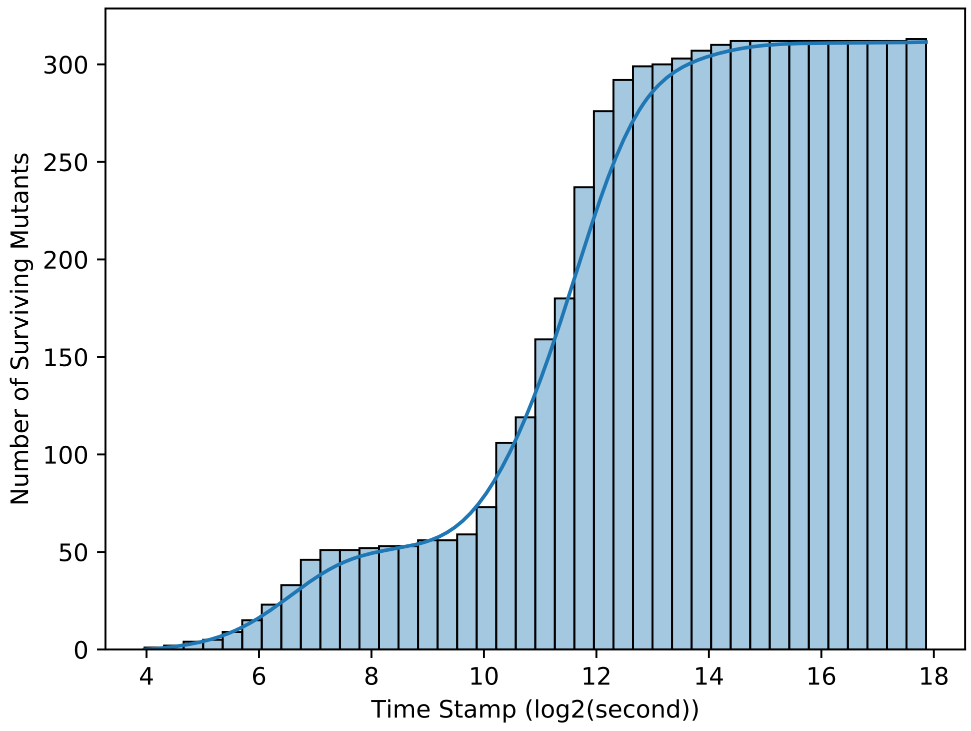

Figure 4.8: Seed distribution

Chapter 5

Discussion and Future Work

5.1 Mutating SPEC.

FAST finds gaps in the SPEC by keeping the SPEC intact and mutating the CODE. Symmetrically, and also proved in theory [101, 102], we can find the same gaps by keeping the CODE unmodified and mutating the SPEC. In the example shown in Figure 3.2, the gap in the SPEC can be revealed by mutating the SPEC into the following snippet.

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

In fact, as discussed in §4.2.3, the initial works on applying mutation testing into a formal verification context mainly focuses on mutating the expressions in the SPEC [28, 89, 143].

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).

5.2 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 [14, 32, 67, 110], 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.

5.3 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.

5.4 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 §4.3, 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.

5.5 Formal Specifications Beyond Verification

The provable way of programming extends beyond providing intuitive security assurances.

In this thesis, we have conducted a systematic analysis of the false assurance problem inherent in formal verification, which might lead to the impression that formal verification tends to be error-prone, and the problems are ubiquitous. It is important to recognize that all programs possess specifications. Even in the absence of formal specifications, informal formats such as documentation and program requirements still exist. The rationale for adopting formal specifications is twofold: they are not only a fundamental component of the formal reasoning process but also impose a disciplined approach, encouraging programmers to aim for more comprehensive and precise methodologies.

5.6 The Misconception of "Formally Verified" Claims

Many projects boasting formal verification often describe their code base as having been “formally verified”, which can create a false impression of being error-free. Such claims can be misleading, potentially leading to significant consequences. Unlike the traditional testing approach, the assumptions underpinning the formal verification model, such as constraints on the running environment, parameter selections, components of the trusted computing base, and the extent of code verification, are critical to the completeness and effectiveness of formal verification results. As such, it is imperative that all assumptions are both clarified and justified [137].

5.7 Metrics to Measure Formal Verification

Building upon the previous point, merely enumerating assumptions in formal verification is insufficient. The quality of the verification, e.g., the sufficiency of the properties in the specification, is unknown. While traditional testing approaches allow for straightforward coverage assessment through source code instrumentation, formal verification presents challenges in both defining coverage metrics and engaging interactively with the verification process to obtain intermediate data. Some studies have utilized coverage metrics to gauge specification completeness and the extent of system behavior coverage. However, these approaches are limited to certain senarios. Coverage metrics are mostly used in simulation-based verifications [173], and are also later adopted by model checking [36]. Mutation testing is another approach that is used to test the robustness of the formal verification model. The deficiency of mutation testing approach is that it tends to be computationally expensive, and it is also difficult to guide the mutation testing process to focus on the portions of a program which are necessary to be checked. Ghassabani et al [73] proposed using Inductive Validity Cores (IVCs), which determine a minimal set of model elements necessary to estabilish a proof.

Chapter 6

Conclusion

In this thesis, we first generalized the formal verification process, within which we identified and analyzed specific stages prone to false assurance. We provided a systematization of knowledge on the false assurance observations at these different stages, together with a discussion on the existing defense mechanisms.

Focusing on the false assurance problem in formal specification, 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] *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL)*, Portland, OR, January 2002.

[2] Why is formal methods necessary. https://shemesh.larc.nasa.gov/fm/fm-why-new.html, 2016.

[3] Certora prover. https://certora.com, 2023.

[4] Infer - static analysis tool. https://fbinfer.com/, 2023.

[5] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[6] Zaher S Andraus, Mark H Liffiton, and Karem A Sakallah. Reveal: A formal verification tool for verilog designs. In *Proceedings of the 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, Doha, Qatar, November 2008.

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

[8] 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.

[9] Jeremy Avigad, Leonardo De Moura, and Soonho Kong. Theorem proving in lean. *Online: https://leanprover. github. io/theorem_ proving_ in_ lean/theorem_ proving_ in_ lean. pdf*, 2021.

[10] 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.

[11] Danial Nikbin Azmoudeh and Yvan Labiche. Analysis of mutation operators for fsm testing. In *Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Dublin, Irelands, April 2023.

[12] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. Smt-based translation validation for machine learning compiler. Haifa, Israel, August 2022.

[13] Haniel Barbosa, Chantal Keller, Andrew Reynolds, Arjun Viswanathan, Cesare Tinelli, and Clark Barrett. An interactive smt tactic in coq using abductive reasoning. In *Proceedings of the 39th International Conference on Logic Programming (ICLP)*, London, UK, August 2023.

[14] 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.

[15] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, Marseille, France, March 2004.

[16] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, July 2011.

[17] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an ssa-based middle-end for compcert. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(1):1–35, 2014.

[18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 1–33, 2018.

[19] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, et al. The dogged pursuit of bug-free c programs: the frama-c software analysis platform. *Communications of the ACM*, 64(8):56–68, 2021.

[20] 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.

[21] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. Stringfuzz: A fuzzer for string solvers. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, Oxford, UK, July 2018.

[22] 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.

[23] 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.

[24] Hamza Bourbouh, Marie Farrell, Anastasia Mavridou, Irfan Sljivo, Guillaume Brat, Louise A Dennis, and Michael Fisher. Integrating formal verification and assurance: an inspection rover case study. In *Proceedings of the 13th NASA Formal Methods Symposium (NFM)*, Virtual, May 2021.

[25] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5, 2009.

[26] Sebastian Buchwald. Optgen: A generator for local optimizations. In *Proceedings of the 24th International Conference on Compiler Construction (CC)*, London, UK, April 2015.

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

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

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

[30] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *Proceedings of the 42th International Conference on Software Engineering (ICSE)*, Seoul, South Korea, October 2020.

[31] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, December 2008.

[32] 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.

[33] Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an llvm fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO)*, Austin, TX, February 2017.

[34] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 24th European Software Engineering Conference (ESEC) / 27th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, New York, NY, August 2019.

[35] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016.

[36] Hana Chockler, Orna Kupferman, and Moshe Y Vardi. Coverage metrics for formal verification. In *Correct Hardware Design and Verification Methods: 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003. Proceedings 12*, pages 111–125. Springer, 2003.

[37] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *Proceedings of the 11st International Conference on Computer Aided Verification (CAV)*, Trento, Italy, July 1999.

[38] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.

[39] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[40] Byron Cook. Formal reasoning about the security of amazon web services. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, Oxford, UK, July 2018.

[41] 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.

[42] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *Proceedings of the 32nd International Conference on Software Engineering and Formal Methods (SEFM)*, Thessaloniki, Greece, October 2012.

[43] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[44] 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.

[45] 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.

[46] Jean Paul Degabriele, Kenny Paterson, and Gaven Watson. Provable security in the real world. May 2010.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, May–June 2016.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th International Conference on Embedded Software (EMSOFT)*, Atlanta, GA, October 2008.

[59] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity*, 5(1):18, 2022.

[60] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J Reynolds, and Cesare Tinelli. Extending smtcoq, a certified checker for smt. *arXiv preprint arXiv:1606.05947*, 2016.

[61] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. Smtcoq: A plug-in for integrating smt solvers into coq. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)*, Heidelberg, Germany, July 2017.

[62] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable ltl model checker. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, Saint Petersburg, Russia, July 2013.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, June 2002.

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

[71] Gordon Fraser and Franz Wotawa. Using model-checkers to generate and analyze property relevant test-cases. *Software Quality Journal*, 16:161–183, 2008.

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

[73] Elaheh Ghassabani, Andrew Gacek, Michael W Whalen, Mats PE Heimdahl, and Lucas Wagner. Proof-based coverage metrics for formal verification. In *Proceedings of the 32th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, IL, October 2017.

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

[75] 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.

[76] 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.

[77] Reiner Hähnle and Marieke Huisman. Deductive software verification: from pen-and-paper proofs to industrial tools. *Computing and Software Science: State of the Art and Perspectives*, pages 345–373, 2019.

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

[79] Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI global, 2015.

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

[81] Gerard J Holzmann. Software model checking with spin. *Advances in Computers*, 65:77–108, 2005.

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

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

[84] 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.

[85] Matthew S Jaffe, Nancy G Leveson, Mats Heimdahl, and Bonnie Melhart. Software requirements analysis for real-time process-control systems. 1990.

[86] 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.

[87] Nathan T Jessurun, Olivia P Paradis, Mark Tehranipoor, and Navid Asadizanjani. Shade: Automated refinement of pcb component estimates using detected shadows. In *Proceedings of the 2020 IEEE Physical Assurance and Inspection of Electronics*, Washington, DC, July 2020.

[88] Ru Ji and Meng Xu. Finding Specification Blind Spots via Fuzz Testing. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 5 2023.

[89] 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.

[90] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. January 2017.

[91] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, et al. Crellvm: verified credible compilation for llvm. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, June 2018.

[92] Chantal Keller. Smtcoq: Mixing automatic and interactive proof technologies. *Proof Technology in Mathematics Research and Teaching*, pages 73–90, 2019.

[93] 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.

[94] 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.

[95] Wilayat Khan, Zhe Hou, David Sanan, Jamel Nebhen, Yang Liu, and Alwen Tiu. An executable formal model of the vhdl in isabelle/hol. *arXiv preprint arXiv:2202.04192*, 2022.

[96] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[97] 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.

[98] Neal Koblitz and Alfred Menezes. Critical perspectives on provable security: Fifteen years of" another look" papers. *Cryptology ePrint Archive*, 2019.

[99] Neal Koblitz and Alfred J Menezes. Another look at" provable security". *Journal of Cryptology*, 20:3–37, 2007.

[100] Moez Krichen. A survey on formal verification and validation techniques for internet of things. *Applied Sciences*, 13(14):8122, 2023.

[101] 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.

[102] 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.

[103] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[104] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. June 2014.

[105] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. July 2015.

[106] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Baltimore, MD, July 2015.

[107] Juneyoung Lee, Chung-Kil Hur, and Nuno P Lopes. Aliveinlean: a verified llvm peephole optimization verifier. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV)*, New York, NY, July 2019.

[108] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P Lopes. An smt encoding of llvm's memory model for bounded translation validation. Los Angeles, CA, July 2021.

[109] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. Taming undefined behavior in llvm. June 2017.

[110] 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.

[111] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[112] 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.

[113] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[114] Liyi Li and Elsa L Gunter. K-llvm: a relatively complete semantics of llvm ir. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[115] 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.

[116] Bin Lin, Jinchao Chen, and Fei Xie. Selective concolic testing for hardware trojan detection in behavioral systemc designs. In *Proceedings of the 2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Grenoble, France, March 2020.

[117] Bin Lin and Fei Xie. A systematic investigation of state-of-the-art systemc verification. *Journal of Circuits, Systems and Computers*, 29(15):2030013, 2020.

[118] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, Genoa, Italy, September 2020.

[119] Yang Liu, Jun Sun, and Jin Song Dong. Pat 3: An extensible architecture for building multi-domain model checkers. In *ISSRE*, pages 190–199, 2011.

[120] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. September 2020.

[121] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 2020 ACM*

SIGPLAN Conference on Programming Language Design and Implementation (PLDI), New York, NY, June 2021.

[122] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Portland, OR, June 2015.

[123] Nuno P Lopes and José Monteiro. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer*, 18:359–374, 2016.

[124] 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.

[125] 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.

[126] José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra 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.

[127] 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.

[128] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 701–712, 2020.

[129] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. Protrr: Principled yet optimal in-dram target row refresh. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[130] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. Rega: Scalable rowhammer mitigation with refresh-generating activations. In *Proceedings of*

the 44th IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, May 2023.

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

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

[133] 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.

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

[135] 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.

[136] 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.

[137] 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.

[138] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: from general purpose to a proof of information flow enforcement. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[139] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.

[140] 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.

[141] 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.

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

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

[144] Samir Ouchani, Otmane Aït Mohamed, and Mourad Debbabi. A formal verification framework for bluespec system verilog. In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, pages 1–7. IEEE, 2013.

[145] 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.

[146] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, July 2019.

[147] 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.

[148] 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.

[149] 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.

[150] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet lightweight row hammer protection. In *Proceedings of the 53th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Athens, Greece, October 2020.

[151] Alexandre Petrenko, Omer Nguena Timo, and S Ramesh. Multiple mutation testing from fsm. In *Formal Techniques for Distributed Objects, Components, and Systems:*

36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings 36, pages 222–238. Springer, 2016.

[152] 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.

[153] 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.

[154] Ratish J Punnoose, Robert C Armstrong, Matthew H Wong, and Mayo Jackson. Survey of existing tools for formal verification. Technical report, Sandia National Lab.(SNL-CA), Livermore, CA (United States), 2014.

[155] 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.

[156] Grigore Rosu. K: A rewriting-based framework for computations–preliminary version–. 2007.

[157] Grigore Roșu and Traian Florin Șerbănută. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[158] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.

[159] 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.

[160] 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.

[161] 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.

[162] Christopher Schwaab and Jeremy G Siek. Modular type-safety proofs in agda. In *Proceedings of the 7th ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV)*, Rome, Italy, January 2013.

[163] Joseph Scott, Trishal Sudula, Hammad Rehman, Federico Mora, and Vijay Ganesh. Banditfuzz: Fuzzing smt solvers with multi-agent reinforcement learning. In *International Symposium on Formal Methods*, pages 103–121. Springer, 2021.

[164] Traian Florin Şerbănuţă and Grigore Roşu. K-maude: A rewriting based tool for semantics of programming languages. In *Rewriting Logic and Its Applications: 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers 8*, pages 104–122. Springer, 2010.

[165] M Siegel, A Pnueli, and E Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lisbon, Portugal, March–April 1998.

[166] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. January 2019.

[167] 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.

[168] Chengnian Sun, Vu Le, and Zhendong Su. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, May–June 2016.

[169] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 31th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Amsterdam, Netherlands, September 2020.

[170] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrücken Germany, July 2016.

[171] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, Grenoble, France, July 2009.

[172] 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.

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

[174] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. June 2010.

[175] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, August 2022.

[176] Bryan Turner. Random c program generator. *Retrieved from*, 2007.

[177] 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.

[178] 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.

[179] Dmitry Vyukov. Syzkaller, 2015.

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

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

[182] 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.

[183] 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.

[184] Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. September 2020.

[185] 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.

[186] 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.

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

[188] 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.

[189] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[190] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. Declarative static analysis for multilingual programs using codeql. *Software: Practice and Experience*, 2023.

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

[192] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, June 2012.

[193] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. Towards better semantics exploration for browser fuzzing. October 2023.

[194] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. {TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, online, July 2021.

APPENDICES

Appendix A

Paper selection criteria

Security	Software Engineering	Programming Language	Formal Verification
Usenix	ICSE	POPL	CAV
NDSS	ASE	PLDI	FM
S&P	FSE/ESEC	OOPSLA	
CCS	ISSTA		

Table A.1: Conferences selection in this thesis

We search for relevant papers that conform to the false assurance problem detailed in §3. Specifically, our initial paper screening is based on the conferences listed in Table A.1, after 2016. Some exceptional papers do not match these two conditions but still got listed in Table 3.2 for some special contribution. We follow the citation trees from the initial set of papers for additional relevant works.