Translation Validation: Case Studies in OCaml-to-PVS and x86 Disassembly

Xiaoxin An

Preliminary Exam

Doctor of Philosophy

in

Computer Engineering

Binoy Ravindran, Chair

Thidapat Chantem

Haibo Zeng

Freek Verbeek

Dongyoon Lee

May 12, 2020

Blacksburg, Virginia

Keywords: Translation Validation, Soundness, Random Testing, Symbolic Execution, Verification, Disassembly, x86/64.

Translation Validation: Case Studies in OCaml-to-PVS and x86 Disassembly

Xiaoxin An

(ABSTRACT)

Translations from one language to another are ubiquitous at different abstraction levels of computer systems including compilation, assembly, and disassembly. Translation's wide application calls for high reliability of the translation process. A key challenge is validating whether a translation is sound. Both the definition of soundness and the validation method heavily depend on the context: what is the source, and what is the destination language.

The literature presents many techniques for validating translation soundness. At one end of the spectrum is testing techniques. They execute the target program under a range of inputs and verify that the program output satisfies expected behaviors. Well-defined semantics are not necessary, however, the destination language is required to have executable semantics. At the other end of the spectrum are techniques such as refinement proofs and formally verified translators. In the former, the destination language is translated into an abstract model, which is then formally verified to establish a refinement relationship with the source language. In the latter, the translator, during construction, is formally verified to establish a conformance relationship between programs written in the source and destination languages. These techniques, however, require that the languages have well-defined semantics. The selection of the validation method is usually based upon the time consumption, reliability, and characteristics of source and destination languages.

This dissertation presents two research contributions for verifying translation soundness. The first contribution targets the case where the destination language is non-executable. We present a test-and-proof methodology called OCaml-to-PVS Equivalence Validation (or OPEV) that validates a translation between the OCaml programming language and the PVS formal verification language. Since OCaml incorporates many external libraries whose source codes are inaccessible, we cannot construct a formal model for OCaml programs calling these external functions. Thus, OPEV uses a test-and-proof technique rather than refinement proofs. OPEV accepts an OCaml program and a corresponding PVS specification as input and generates large-scale test cases, which are directly executed on the OCaml

program as well as used to generate PVS test lemmas. OPEV incorporates an intermediate type system that captures a large subset of OCaml types, and employs a variety of rules to generate test cases for each type. We demonstrate OPEV on two case studies: an OCaml-to-PVS translator and a Sail-to-PVS parser. OPEV generated and proved 458,247 test lemmas for these case studies and detected 11 errors.

The dissertation's second contribution targets the case where the source language does not have well-defined semantics. Disassembly is a translation with a raw binary as the source language and a mapping of memory addresses to assembly instructions as the destination language. Disassembly is an undecidable problem: it is theoretically impossible to accurately distinguish instructions from raw data and to predict all the branch decisions. We present a formal definition of the soundness of a disassembly process. A tool called Disassembly Soundness Validation (or DSV) is presented that takes as input 1.) a raw binary and 2.) the output of some black-box disassembler such as objdump. DSV verifies whether each disassembled instruction is correct and reachable from the binary's entry point. We demonstrate DSV on a set of elaborately designed micro-benchmarks that is inspired by the GNU Coreutils library.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

CFG control flow graph

DSV disassembly soundness validation

EGT execution-generated testing

ISA instruction set architecture

OPEV OCaml-PVS equivalence validation

Chapter 1

Introduction

A multitude of computer languages has been developed ever since the invention of the computer system. These languages serve as an indispensable element in modern computer system design, implementation, and usage. Machine instructions, which have no well-formed semantics and are represented as byte sequences, indicate the operation that the computer is performing during execution. Since the machine language is composed of byte sequences and is hard to understand, researchers developed assembly language to represent the machine instructions in a human-readable way. The assembly language is classified as a low-level programming language since the assembly instruction is one-to-one mapped to machine instruction. Correspondingly, programming languages with high-level features such as modularity, scoping, data structures, and functions are called high-level programming languages.

The high-level programming languages are divided into different classes and have various applications. Functional programming languages, such as OCaml [53] and Haskell [76], support structured programming where the major elements are pure functions. Meanwhile, object-oriented programming language, such as C++ and Java, assists programmers to organize code into various classes and objects. Besides, there are logic programming languages, such as Prolog [28], and formal verification languages, such as HOL4 [71], Isabelle/HOL [60], Coq [16], and PVS (short for Prototype Verification System) [61]. The languages belonging to different categories have different natures and semantics, which leads to a deep gap between these languages.

At the same time, it is quite common to translate one language to another language. For example, to execute the code written in a high-level programming language, we need to use a compiler to translate the code to an executable file. On the other hand, for conducting various forms of analysis on a binary file, disassemblers are often employed to translate the binary to assembly code, and then further research is conducted on the assembly code, which is common in reverse engineering [21]. Up to this day, there are many commercial and open-source translators that take the responsibilities to translate from one language to another language.

1.1 Motivations

The literature presents numerous examples of translations that take a language *with well-defined semantics* as input and as output a language *without well-formed semantics*. These cases are common for most compilers that translate code written in a high-level language to binary or executable such as GCC [73], LLVM [49], the Java compiler [68], Haskell compiler [76], etc. However, for situations where translation from a functional programming language to a language that enables formal verification, or from a binary to a programming language is desired, this is a challenging problem.

Since language translation plays a key role in computer system design, implementation, and usage, various methods have been used to validate the trustworthiness of the translation. Testing is the most common validation method. As an example, Anthony Fox presented a trustworthy formalization of the ARMv6 [20] instruction-set-architecture (ISA) in HOL4 [31] – a language that allows specification in higher-order logic.[1]. To demonstrate that the model complied with the intended meaning of the ARMv6 chip designers, the formal model in

[1]To be precise, HOL4 is a proof assistant [71] that allows specification and proof in higher-order logic [56].

HOL4 was litmus-tested against the actual physical behavior of hardware that used the same ISA. Later, in [32, 33], this approach was enhanced to be more automatic and was used to formalize other ISAs such as ARMv7 [15], POWER PC [72], MIPS [40], and a subset of x86 [38]. Another example is Lem [7], which was designed to serve as a reusable semantic model of programming languages that was mathematically rigorous [57]. Lem can be translated to OCaml for emulation of testing as well as to Isabelle/HOL [60], Coq [17], HOL4 [71], and other languages. These translations were validated via pre-defined tests written in the Lem language [6].

Other than testing, a more rigorous methodology that can be used to verify the translation is *refinement proving* [45]. This method requires a translation into a formal verification language to generate a formal proof. Meanwhile, the translated model, whether it was generated manually or mechanically, has to comply with the intended meaning of the program under verification. For example, seL4's formal verification used a translation from a subset of C called C_0 into Isabelle/HOL [39, 45]. The conformance relationship was established based on refinement proof that required significant human effort [37]. The approach, over two decades, demonstrated that the formal verification of a complex software of about 10K LOC is possible without sacrificing performance. The approach, however, was meant specifically for seL4 and C_0 [45].

Building up a translator that is formally verified is another method that increases the reliability of the conformance relationship between the source and target languages of the translator. A formally verified translator models the formal semantics of the source and destination languages that are used in each step of the translation. Then it applies formal proofs on semantics preservation [52] to show that the translation is sound. For instance, CompCert [50] is a compiler that was formally verified to compile C programs to assembly code. The compilation guarantees that the assembly code executes with the behavior that

was designated by the original C program [42]. However, the formal proofs of CompCert did not cover the correctness of the formal specifications of C and assembly [52]. In addition, it took six person-years of effort and involved 100,000 lines of Coq code [43].

1.2 Challenges

The validation of a translation between a language with well-defined semantics into a language without well-defined semantics involves a key problem: how to define the *soundness* of the translation. Since the source and destination languages do not have formally-specified behaviors, the conformance relationship between the programs written in these two languages requires an elaborately designed definition. Generally described, the soundness of the translation means that the two programs have the same behaviors and their execution results with similar inputs are equivalent to each other.

The choice of the methodology that is employed to validate the translation is another problem. Though refinement proof is a rigorous method to validate the translation, it is challenging to build a formal model for both source and target languages since in most cases the semantics of the two languages cannot be mapped to each other one-to-one. Besides, if one language does not have well-formed semantics, extracting a formal model for it is another challenge. In such a situation, random testing [29] is also a choice since well-designed random testing cases could cover most of the situations and assist users to find inconsistencies.

Even though random testing is a straightforward technique and has been widely studied, there are situations where it is not applicable because the destination language is not executable. For example, some formal verification languages, such as PVS [61], do support specifications that are not executable. To verify certain properties on the destination language, researchers need to employ specific features of the formal verification language and

build up the soundness validation method.

1.3 Dissertation Contributions

State-of-the-art techniques for translation validation exist in many pieces of literature. Refinement proofs, which are applied to seL4 OS kernel [46], require great time and human efforts to construct the formal model and prove the conformance relationship. Verified translator, such as CompCert [52], also requires considerable manpower and time to formally verify certain properties for the translator. Testing is widely applied in the software industry and is easy to implement. However, if one language of the translation supports non-executable semantics, such as PVS, the pure testing technique is not applicable. Meanwhile, few related works have been proposed for the soundness validation of the disassembly process. Paleari et al. [64] introduced a validation method based on differential analysis to test different x86 disassemblers. Another validation methodology for the disassembly process was developed by Andriesse et al. in [14] which studied the accuracy and various properties on nine disassemblers using ground truth about ELF [26] information collected by an LLVM [49] analysis and on DWARF [30] v3 debugging information.

This dissertation presents two research contributions. The first contribution is a technique for validating the equivalence relationship within an OCaml-to-PVS translation. The motivation for validating the OCaml-to-PVS translation lies in the requirement for translation from the Sail language to PVS. Sail language [36], which is a first-order imperative language, has been used to describe the semantics of ISAs such as x86, ARM, RISC-V, and PowerPC [36]. Sail specifications of many of these ISAs have been used for type-checking and test-case generation, translated into executable emulators, and lifted into theorem-proving languages for rigorous reasoning [36]. While translators from Sail to theorem provers such

as Isabelle/HOL, HOL4, and Coq exist [36], one to the PVS [61] does not. We developed a Sail-to-PVS translator to translate the semantics of many ISAs which is modeled in Sail for the benefit of the PVS community. It is critically important that the translation from Sail to PVS is provably correct. We presume that the translation from Sail to OCaml is trustworthy, then we employ the executable feature of OCaml to validate the OCaml-to-PVS translation. If the equivalence between OCaml and PVS is verified, the Sail to PVS translation is validated. Motivated by these concerns, we present a test-and-proof methodology to validate the translation from OCaml to PVS. This validation is challenging since OCaml does not have well-defined semantics while PVS has. Moreover, PVS is a formal verification tool and supports non-executable semantics. We employ specific features of PVS such as subtypes [67], proof checking [62], and batch proving [58] to solve the validation problem.

The dissertation's second contribution is a technique for the soundness validation of the disassembly process. Disassembly is a translation from machine code in a binary to assembly code. The machine code in a binary has no well-defined semantics, whereas assembly code has. We set up a limitation for this problem in that we do not have access to the original assembly code, and thus we do not have the assembly code as the trustbase for this validation. This limitation is motivated in part by application settings where assembly code is wholly or partially unavailable, outdated and decaying build processes and environments that prevent regeneration of assembly, and third-party libraries and tools that are no longer available or backward compatible. This limitation raises the problem of how to determine the trustbase for soundness validation, while requiring that correctness of the disassembly on the binary files must be verified without assembly code. Besides, we focus on the widely used Intel x86 ISA [54], which is another challenge since x86 ISA has a great number of instructions with variable lengths and the documentation related to x86 ISA does not provide the formal specifications for some instructions.

1.3.1 OCaml-to-PVS Equivalence Validation

We present a semi-automatic test-and-proof methodology to validate the translation between two different languages with one of them supporting non-executable semantics. The test-and-proof method combines testing and proving together to validate certain properties, which requires short development cycles and supports validation on formal verification language. Our methodology (Chapter 4), folded into a tool called OPEV, for "OCaml-to-PVS Equivalence Validation", takes an OCaml program and a corresponding PVS implementation as input. From these inputs, OPEV automatically generates large-scale test cases using the generating rules we have developed for the intermediate type system which captures the commonality of OCaml and PVS types. The test cases are directly executed on the OCaml program and also used for constructing a large number of test lemmas on the PVS specification. We represent the test lemmas as the equations with the test cases as the left-hand side and the executing results as the right-hand side. Since the test lemmas are represented as equations that do not hold any higher-order logic, we are able to automatically prove the test lemmas using a specific PVS feature called proof strategies [58]. The results of the proofs are then employed to establish equivalence. Fig. 1.1 illustrates OPEV.

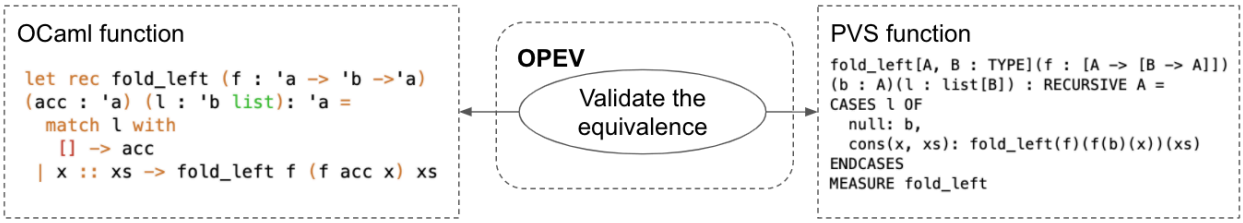

Figure 1.1: Equivalence validation for OCaml and PVS.

We demonstrate OPEV by using it to validate a manually implemented OCaml-to-PVS translation and a Sail-to-PVS parser (Chapter 5) that we manually developed. The Sail-to-PVS parser includes 2,763 LOC and was used to translate 7,542 LOC of Lem code to 10,990

LOC of PVS implementation. OPEV generated and proved 458,247 test lemmas for these two case studies, and detected 11 errors (Chapter 5). The development of OPEV took 3 person-months and the effort to develop and validate the translator took 5 person-months.

1.3.2 Disassembly Soundness Validation

We propose a formal definition for the soundness of the disassembly process and prove the validity of the soundness definition to build a grand foundation for the trustworthiness of our work. The soundness definition of the disassembly process illustrates the correctness of the translation from binary to assembly code. That is, the formal definition is basically capturing the reachability of each instruction from entry point and validating the accuracy of the instruction recovery during the disassembly. We implement the soundness definition in a tool called DSV (short for Disassembly Soundness Validation) and employ the tool to validate the soundness of disassembly. Our definition provides a ground truth, which is independent of any specific assembler or disassembler. However, in the implementation of DSV, we have to capture the semantics of the x86 ISA, which we have done manually and have also validated their correctness. Besides, we have developed a bottom semantics to fill in the blanks caused by the unimplemented instruction semantics and a trace-back model to help to solve the indirect jump address problem.

As illustrated in Fig. 1.2, DSV takes the binary file and the generated assembly code – both for the x86 architecture – as input and validates whether the disassembly is sound or unsound. We apply DSV on a set of micro-benchmarks that are inspired by features in the GNU Coreutils library [4]. Since our work does not require that the binary file has a source assembly code, DSV can validate legacy or commercial binary software without source code.

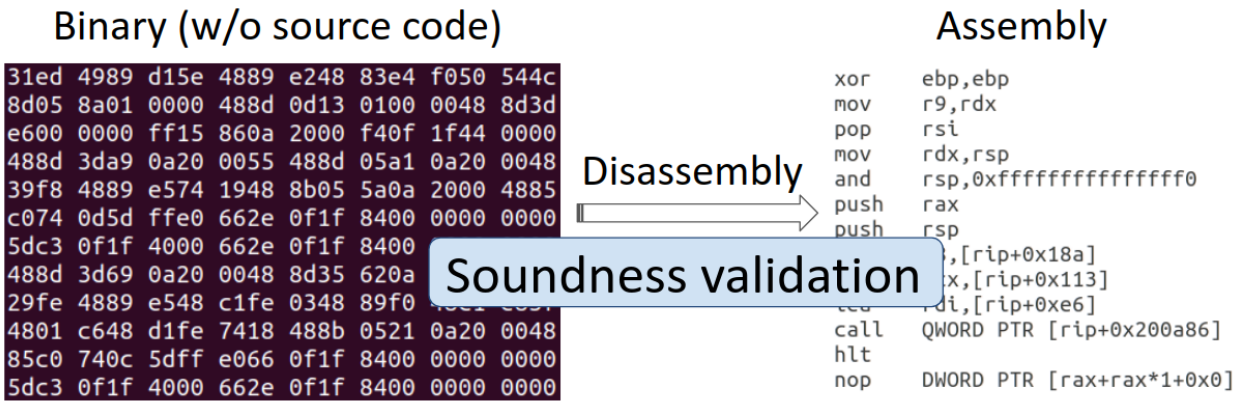

Figure 1.2: Soundness validation for disassembly.

1.4 Dissertation Organization

Chapter 2 presents the background of symbolic execution, formal verification, and model checking. Chapter 3 presents past and related work. The OCaml-to-PVS equivalence validation work is presented in Chapter 4. We introduce the proof automation of PVS and the corresponding case studies of OPEV in Chapter 5. We present the soundness definition of disassembly in Chapter 6 and the implementation of the disassembly soundness tool in Chapter 7. The case studies of the DSV tool is demonstrated in Chapter 8. The dissertation concludes in Chapter 9, which also presents the proposed post-preliminary exam work.

Chapter 2

Background

In our work, we refer to different kinds of formal methods and apply testing, semi-automatic proofs, and symbolic execution to build up our tools. We introduce the relevant background information in detail in the following sections. Section 2.1 introduces the fundamentals and limitations of symbolic execution. We present a formal verification technique and its application in a real system in Section 2.2. Section 2.3 demonstrates the principles of model checking and some essential work that is implemented using a model checking technique.

2.1 Symbolic Execution

The idea of symbolic execution was introduced in 1976 by J.C. King [44]. Once the idea came out, it was widely applied in software analysis, model checking, software testing, etc. In the original design of symbolic execution, the inputs are symbolic value, and the program executes on the symbolic inputs to explore as many execution paths as possible to check certain properties of the program. Symbolic execution infers input classes instead of individual input values. In more detail, each value that cannot be determined through static analysis of the code, such as the actual parameter of the function or the result of a system call that reads data from the stream, is represented by the symbol value. By evaluating all the generated execution path, certain properties are verified.

The major limitation of symbolic execution is the path explosion problem. Take the code

in Listing 2.1 as an example, our target is to verify the assertion at line 8. With concrete input a, we can execute the program and verify that the assertion is *true* under concrete input value. However, we cannot declare that this assertion is *true* in any case. If we apply symbolic execution on the code, the symbolic execution generates an unlimited number of execution paths since the code contains loops and recursion, and the termination condition is a symbol.

Listing 2.1: An algorithm to calculate square using addition.

```
void foo(int a) {
    int sum = 0;
    int b = 1;
    for (int i = 0; i < a; i++) {
        sum += b;
        b += 2;
    }
    assert(sum == a * a);
}
```

Moreover, if the symbolic path constraint is not solvable or cannot be solved efficiently, then input cannot be generated. Assuming that the adopted solver cannot solve the constraint generated in the execution path, then the symbolic execution will fail, that is, symbolic execution will not be able to generate any input for the program or verify the properties that are required in the program. For the unsolvable path constraints, techniques, such as concolic testing [69], are developed to solve the problem. Moreover, to reduce the number of generated paths, algorithms that eliminate infeasible paths during the symbolic execution are developed [13].

2.2 Formal Verification

Formal verification technique in a computer system is to prove mathematical theorems using assistant tools. The proving process is based on reasoning logic, such as temporal logic and natural deduction rules which describe logical reasoning using inference rules. Some theorems or lemmas we need to prove are in the form of propositions, which take the value of true or false. These propositions are deduced from various premises by implementing different inference rules.

Generally, it takes three steps to formally verify a practical system. First, formulating the specification of the model of the system using certain language in theorem provers. Since the functional languages used in theorem provers are different from the programming languages, the first step of construction takes a long time and great effort. For example, in the verification of seL4 [46], it took the working team 9 person-years to build up the formal frameworks and tools, which occupied almost half of the whole time spent on the project. Second, proving the correctness and soundness of the specification. Due to the existence of concurrency and indeterminacy in a real-time system, this part also requires special expertise and considerable endeavor. The third step is to implement a practical system that meets the specification and verify the refinement relationship between the specification and the real system. For some theorem provers which have integrated code generator, such as Isabelle/HOL theorem prover [60], the implementation can be fulfilled automatically.

Generally, the theorem-proving language is dissimilar from the languages that are used to implement real software systems. Therefore, it is impossible to obtain the theorem-proving model directly from the real system. The modeling process usually takes a long time. Besides, proving the theorems representing the properties of the system is also time-consuming and skill-requiring, which prevents the application of theorem-proving techniques in some

system verification. However, if the theorem has been successfully proved, the corresponding property regarding the real system is highly trustworthy.

Many theorem provers, such as Coq [16], HOL4 [35], Isabelle/HOL [65], and PVS [61], are developed using distinct languages, such as OCaml, Standard ML, and Common Lisp. Although their implementation methods are different and the reasoning logic is not necessarily identical, these theorem provers have already been applied to analyze different systems. For example, Isabelle/HOL has been used to verify seL4 [45], an OS microkernel. Coq has been applied to analyze Verdi [81], a distributed system, and Compcert [43], a C compiler.

The advantages of these theorem-proving tools are that they **prove** the general concepts, rather than verifying them using specified variables, states, and traces, which prevents particular errors due to loss of details. For example, in [78], if the number of processors in a multiprocessor system is parameterized, which means any number of processors is acceptable is the system, then it is infeasible to apply model checkers to verify the system. In contrast, theorem provers are applicable for such a parameterization problem since the number of processors does not affect the final results.

Formal verification using interactive theorem provers has been extensively applied to software and hardware systems in recent years, with the increasing requirement of system correctness and soundness. In the software field, Compcert, a C compiler, has been verified using Coq theorem prover in [51]; and Ridge et al. presented a model of the behaviors that are permitted by SibylFS file system in [66]. In the hardware field, Vijayaraghavan et al. modeled, refined, and proved a multiprocessor hardware system that consisted of a parameterized number of processors and parameterized level of cache hierarchies using Coq theorem prover [78], which is a highlight in the verification of hardware system.

Even further, Klein et al. [46] have employed Isabelle/HOL to formally verify seL4 micro-

kernel from the specification of the model to the low-level C implementation, this totally demonstrated the wide range of application of theorem provers in software verification. There exist many challenges in the verification of operating systems, such as the large-scale code base of the kernel, the abstract model of the real implementation, and the proofs that are required for the refinement relation between the abstract model and the real implementation. seL4 [45] provides high-level assurance of functional correctness of an OS kernel in L4 family using formal proofs. To fill the gap between a real kernel and its abstract model, seL4 took a methodology that started from a medium-level prototype written in Haskell. The intermediate specification was then directly translated to a formal abstract specification and was manually re-implemented to a C implementation. seL4 was tested with OKL4 2.1 on a specific platform. The performance of seL4 was approaching the other optimized L4 kernels, which indicated that the formally verified kernel can also achieve high performance. The complete functional correctness of seL4 is verified. Besides, the researchers assumed the correctness of the C compiler and the real hardware, which are the trustbase in the verification. There are some limitations during the verification of seL4. For instance, seL4 only allows a large subset of C99 language. Besides, they spent 20 person-years on the highly-assured kernel with almost 10K LOC.

2.3 Model Checking

Model checking is an automatic verification technique that searches the finite state space of the system model to check whether the system's behavior satisfies predictive properties. A formal model-checking approach is to employ a particular language to construct the model of the system, describe the specifications of the requirements in the form of formulas, and analyze whether the model conforms to the specifications.

Model checkers analyze different properties according to the specific requirements of a certain system. First is the correctness of the model, which means that the modeling process should conform to the rule of certain model-checking language and incur no error in any traces. Most properties of the model are generally divided into two kinds: safety property, which means that nothing bad would happen, and liveness property, such as the termination which can be verified using temporal logic. Besides, different model checkers analyze distinct properties. For example, CBMC [47] could verify array bounds (buffer overflows), pointer safety, exceptions, and user-specified assertions in C code; and TLC [48] checks the specification of some simple system constructed using TLA+ or PlusCal language.

Explicit-state model checkers, such as Murphi [55], TLC [48], or SPIN [41], are only able to handle finite-state models. They iterate over all the behaviors and analyze the model. If the number of states is too large or even infinite, explicit-state model checkers are not capable. Symbolic model-checking tools including SMART [22] and NuSMV [23] have better performance to analyze this kind of complex system with infinite states. However, to apply symbolic model-checking techniques, the states of the model have to be symbolized and classified into various finite sets, and traces be translated into transitions between sets.

As the technique develops, automation becomes a key requirement in model checking. For example, CBMC [47] is applied to test C program and verify predefined properties automatically. However, in some cases, because of the undecidability of the problem, such as whether a C program would terminate or not, and infinite state space, it is impossible to apply automatic proving all of the time. Handwork is still necessary in such cases. Besides, even though model checkers can be employed to prove some rather complex system, it is still impossible for them to verify a full operating system like seL4, since real OS has too many features and indeterminacy, and state explosion is still a great challenge in model checking.

2.3.1 Application of Model Checking

Although a model is needed to be constructed for the system, the automation property of the model check lowers the threshold of model checking technique. Model checking technique is widely applied to verify many software and hardware systems.

Linux Virtual File System [34]

This paper introduces a model of the Linux Virtual File System (VFS) and shows how to verify the validation of the model. The model is extracted from C source code of the implementation of VFS together with some manually inserted code. Then, SPIN [41] and SMART [22], two different model-checkers, are respectively applied for simulating and verifying the model.

The process of constructing a VFS model is to extract the activities from the real implementation of VFS and articulate them in an abstract method. Because of some specific elements, such as dynamic memory allocation, macros, and inlined assembly, the modeling process cannot be executed totally automatically. Thus, Kernel Function Trace tool is selected to get traces from the executions of a Linux kernel, and manually examination is also adopted to assemble an abstract VFS model in C language.

The simulation of the model is implemented using SPIN model-checking technique for the following reasons. First, Promela language, which is used in the checking process of SPIN, is quite similar to the C language that is employed in the model. Second, SPIN has great simulation competences and accepts assertions inserted while running. Thus, SPIN model can be used to simulate the C model and to detect model errors via simulation of various system calls using SPIN. However, due to the broad state space along with the concurrency in VFS implementations, SPIN is not capable to verify the model.

Therefore, SMART, a symbolic model checker, is introduced to verify the model. The verification using SMART is based on Petri nets, thus the VFS model is translated into a Petri net and then analyzed using SMART. In the new model, various VFS variables are parameterized and symbolized as Petri net node, and calls are depicted as transitions. The main properties of the VFS model which are verified using SMART checker are deadlock-freedom and data-integrity which includes three elements: allocation, reference, and structural properties.

Hypervisor Framework [77]

In this paper, the authors present a new hypervisor framework called XMHF (eXtensible and Modular Hypervisor Framework) which supports further extensions and preserves essential memory-security property. XMHF is limited to support only a single guest (other hypervisors or operating systems) and sequential execution, and it holds certain properties such as Modularity, Atomicity, and Initialization Validity. In the model-design procedure, all the properties should be realized to ensure memory integrity, which is proved by illustrating that system invariants, referring to memory integrity, would resist under all circumstances.

After the fundamental proofs of system security, it is verified that the extensions based on this framework are still correct in memory integrity. This part is automatically analyzed since the extended hypervisor is developed over the framework and also has specific properties, which can be verified using CBMC [47]. This framework is evaluated by comparing with other general hypervisors, and the evaluation results show that the performance of XHMF is as great as other popular hypervisors.

The verification of the structure is implemented by CBMC, a model-checking technique, and the majority part of the code is analyzed automatically, except for a small part regarding concurrency and loops over page tables. The code is verified directly due to the functionality

of CBMC, which eliminates any inaccessible code and unfolds other codes. Because of the simplification of the framework (single guest and sequential execution), only a minor part of the code, including concurrency and unboundedness of the code, needs manual verification, which immensely reduced the handwork.

Chapter 3

Past and Related Work

In this chapter, we present OPEV's related work in Section 3.1. Section 3.2 introduces the linear and recursive disassembly and some work regarding translation validation of disassembly process.

3.1 Translation Validation

Significant literature exists in translation validation. Due to space constraints, our discussion is not meant to be comprehensive; we only discuss the most relevant and closest efforts to ours.

CompCert [43, 52] uses a *formally verified compiler* to establish the correctness of compilation from a subset of C to PowerPC, ARM, RISC-V, or x86 assembly code. The compilation guarantees that the assembly code executes with the behavior that was designated by the original C program [42]. However, the formal proofs of CompCert did not cover the correctness of the formal specifications of C and assembly [52]. In addition, it took six person-years of effort and involved 100,000 lines of Coq code [43].

In [70], the authors show that the seL4 source code [12] and its binary code have the same behavior. The translation validation, in this case, relies on a *refinement proof*. A refinement proof is possible here due to formal semantics that was created for both the source and

target languages. However, the semantics of Sail and PVS cannot be mapped to each other one-to-one. Besides, refinement proofs, in general, are labor-expensive due to the significant human intervention that is necessary. The seL4 refinement proof [45] took 8 person-years; the seL4 total verification effort [45] is more significant and took ∼20 person-years.

In contrast with *compiler verification* and *refinement proofs*, OPEV is a light-weight approach for the validation of a translation from a high-level language into a theorem prover using *random testing*. OPEV is therefore significantly less labor-expensive. In addition, OPEV allows non-executable specifications and proofs for generic theorems after translating the code for further verification. The comparison between OPEV and other translation validation methodologies are illustrated in Table 3.1.

Table 3.1: OPEV methodology vs. other translation validation techniques.

Feature	Sel4 [45]	CompCert [52]	OPEV
Methodology	Refinement Proof	Compiler Verification	Random Testing
Total LOC	+ 210K	100K	23,615
Target LOC	10K	NA	9,000
Time requirement	8 person years	6 person years	1.5 person years

OPEV also differs from some other test-based light-weight verification techniques. For instance, Haskell's QuickCheck mechanism [25] is designed to aid in the verification of properties of a given function. The tests are randomly generated until either a counterexample is discovered in a given domain or a preset threshold is reached. Likewise, AutoTest for Eiffel [24] checks program annotations based on randomly generated test suites. Similar methods exist for theorem provers. For example, QuickCheck [75] and Nitpick [19] for Coq and Isabelle/HOL uses random testing [79] to support counterexample discovery for a given conjecture these mechanisms work well with executable specifications. OPEV differs from these efforts by its focus on validating the translation into a theorem prover as shown in Table 3.2. Precisely, OPEV aims to increase the trust in the translation process of code into

its formal specification (including the non-executable) based on the random testing. These tests do not attempt to prove or disprove any functional property, but they increase the trust in the formal translation. However, our translation into PVS may allow the user to *verify* properties and specified conjectures for the translated functions using PVS's built-in test-generator [27] to assist proving these properties or reaching a counterexample [63]. But like the other built-in translations, it is restricted to generated PVS's executable specifications from our tool.

For translating non-executable specifications, OPEV allows proofs using pre-designed, automatic proof strategies for translation validation.

Table 3.2: Comparison between OPEV and light-weight formal verification approaches.

Tool	Non-Executable Spec Verification	Translation Validation	Counterexample Search
OPEV	✓	✓	✓
QuickCheck	×	×	✓
Eiffel AutoTest	×	×	✓

The closest work to OPEV is MINERVA [59], which provides a practical approach to produce high assurance software systems using model animation on mirrored implementations for verified algorithms [59]. However, this work is limited to the executable subset of PVS. OPEV can be viewed as complementary to MINERVA when the specification is not executable.

3.2 Disassembly Validation

3.2.1 Linear and Recursive Disassembly

Linear sweep and recursive traversal are the two main techniques behind the binary disassembly process. Objdump, PSI [82], and OllyDbg [8] are typical linear-sweep disassemblers.

These disassemblers process the byte sequences in the binaries sequentially and generate the assembly code based on the decoding of the byte sequence. Linear-sweep disassemblers have superior performance under certain circumstances. For example, as shown in [14], some linear disassemblers fulfill a 100% correctness on SPEC CPU2006 benchmarks generated by gcc and clang. However, these disassemblers have poor performance to handle special situations such as overlapping instructions, obfuscated code, and inline data such as jump tables.

On the other hand, disassemblers such as IDA pro [5], Dyninst [18], angr [74], and Ghidra [2] are implemented using recursive traversal. These disassemblers decode the instructions following the execution of the sequential and branch instructions, recognize the jump addresses, and construct a control flow graph (CFG) for the disassembly process. The recursive disassemblers handle the overlapping instructions and inline data in a trustworthy way, which prevents specific errors that can happen for linear disassemblers. However, the recursive traversal presents a crucial challenge for these disassemblers, which is how to resolve indirect jump addresses. The implementation of jump address resolving algorithms in various disassemblers leads to different performance for these disassemblers.

3.2.2 Soundness Validation

The disadvantages existing in linear sweep and recursive traversal methods lead to disassembly errors for various disassemblers. There are a considerable number of papers studying the problem of how to find disassemble errors. N-version disassembly [64] applied differential analysis to verify the correctness of different x86 disassemblers. The writers used CPU as the trust base and compared the disassembled results. However, the usage of CPU status was not that trustworthy and this paper just checked the correctness of single instruction and randomly generated a binary string as the test cases.

Andriesse et al. [14] checked the false positive and false negative rate for 9 main-stream disassemblers using SPEC CPU2006 and glibc-2.22 as the benchmarks. This paper collected the trust base from LLVM analysis and DWARF debugging information. The researchers gave a comprehensive comparison between different disassemblers on 5 key criteria. Besides, they compared some recent literature related to disassembly using 5 criteria and drew some conclusions regarding corresponding work. Since the researchers employed LLVM and Capstone v3.0.4 to construct the ground truth, the results were limited by the accuracy of these tools.

Wang et al. [80] built up the reassembly process by implementing a tool called Ramblr and took a survey on the false positive and false negative of disassembly on different libraries. The tool Ramblr reassembled binaries to the corresponding assemblies. Besides, Ramblr did not introduce execution overhead and supported optimized binary files. The researchers developed many methods using *angr* framework to handle critical challenges happened during the reassembly process.

Chapter 4

OPEV: OCaml-to-PVS Equivalence Validation

In this chapter, we introduce the OCaml-to-PVS equivalence validation (OPEV) methodology that increases the trust in the translated OCaml code into PVS. The translation can be automatic (for a subset of OCaml) or manual. We present the overall workflow of OPEV methodology in Section 4.1. Section 4.2 introduces the intermediate type system that we developed to incorporate the commonality between OCaml and PVS languages. Then we demonstrate how to generate test cases and test lemmas in Section 4.3. The proofs of the generated test lemmas are shown in Section 4.4.

4.1 OPEV Workflow

Figure 4.1 shows the OPEV workflow. In OPEV, we have designed an intermediate type system, Subsection 4.2, to capture the commonality of OCaml and PVS types, which are restricted to a subset of the complete OCaml and PVS types. OPEV parses the PVS and OCaml sources to construct the intermediate type annotations for each function. With these annotations, OPEV generates random test cases for every OCaml and PVS function. OPEV then runs the OCaml test cases to obtain the test results, translates the OCaml test results to PVS, and constructs PVS test lemmas using the PVS test cases and translated results. The

test lemmas are employed as *test oracles*, which can be automatically verified using manually implemented, generic PVS proof strategies. If the test lemmas are proved to be false, we know that there are mismatches in the OCaml-to-PVS translation. Thus, we investigate the cases and try to detect the reasons. The total codebase of OPEV is 3,783 LOC.

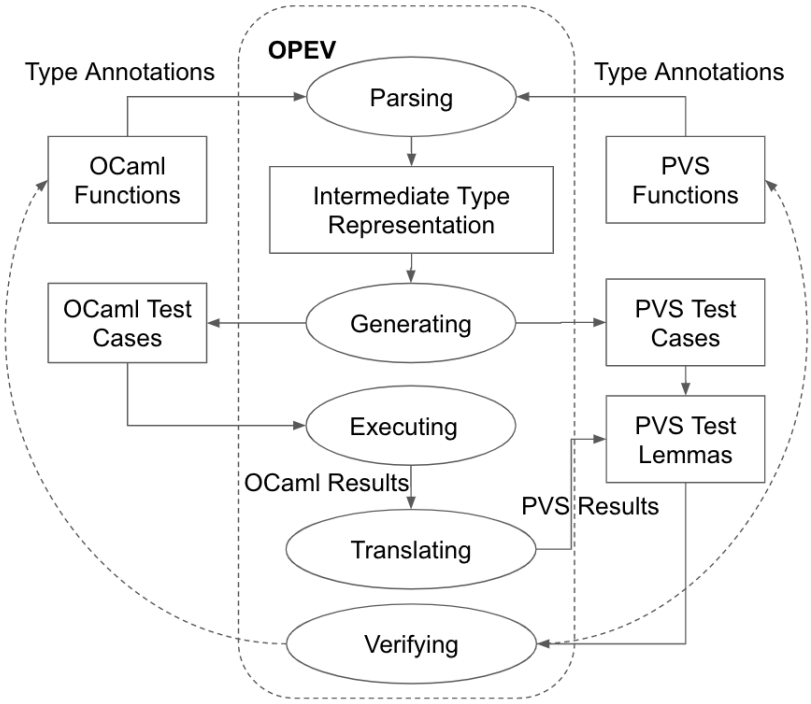

Figure 4.1: The OPEV workflow.

4.1.1 Extensibility

OPEV has already incorporated the semantics of a large subset of OCaml and PVS for automatic test-generation. To ensure that OPEV can be extended to incorporate more types in the future, we represent the generated test cases and testing results in the `string` format to circumvent the real type system of OCaml and PVS.

For example, in Listing 4.1, suppose we randomly generate [1, 6, 8] as the test value for the argument `l` of function `rev`. We then construct a string "let res = rev [1; 6; 8];;" as the

OCaml command and delegate it to the OCaml `Toploop` library to execute the command. The result can be fetched from the `res` variable, which has the value [8; 6; 1]. Then OPEV parses the result according to its type and composes a PVS test lemma, such as `th_rev` in Listing 4.2.

Listing 4.1: A sample PVS reverse function.

```
rev[A:TYPE](l:list[A]) : RECURSIVE list[A] =
CASES l OF
    cons(x, xs): append(rev(xs), cons(x, null))
    ELSE null
ENDCASES
MEASURE length(l)
```

Listing 4.2: A sample of OPEV PVS test lemmas for `rev` function.

```
th_rev: LEMMA rev((: 1, 6, 8 :)) = ((: 8, 6, 1 :))
```

The lemma is also written in the string format. This string-format representation allows us to avoid writing various functions for different argument types and simplifies the extension of OPEV.

4.1.2 Non-Executable Semantics

We construct PVS test lemmas rather than directly executing the PVS test cases because the semantics of some testing functions are non-executable. That is, in PVS, functions with non-executable semantics cannot be executed using the PVS ground evaluator and PVS built-in strategies. For instance, most functions with set-theoretic semantics in PVS are non-executable, including relational specifications, which are represented as predicates on

sets in PVS. For example, the semantics of the function `filter`,

Listing 4.3: A PVS function with non-executable semantics.

```
filter[A:TYPE](p:[A->bool])(s:set[A]):set[A]=
    {x: A | member(x, s) AND p(x)}
```

shown in Listing 4.3, is non-executable. This is because the `filter` function simply describes what kind of elements should be in the result set after the execution of the function but does not specify the steps of how to execute the function in PVS executable syntax. For instance, trying to execute this function directly in PVSio will issue an error message that indicates the `filter` function includes a non ground expression.

4.2 Intermediate Type Classification

To generate tests for OCaml and PVS functions respectively, we have to determine the commonality and difference between the two languages. Therefore, we design an intermediate type system to fill the gap between the type systems of the two languages. Since the types of the two languages cannot be matched with each other one-to-one, we classify the types of the two languages into five different classes and design rules to handle them separately.

OPEV's intermediate type system is categorized into 6 different classes: `PEmpty`, `PBasic`, `PComplex`, `PDef`, `PExt`, and `PSpec`. In this classification, `PEmpty` represents a dummy type that is used as a placeholder to occupy some blank space in the type notation. The existing OCaml types are then grouped according to the remaining five classes. Namely, basic built-in types such as `bool`, `nat`, and `int`; complex data types such as `string`, `tuple`, and `list`;

user-defined types including `datatype`, `record`, and others; external library types; and types requiring special treatment such as `functional` types.

For each intermediate type, we design a generating rule and parsing rule according to the class of the type. Currently, OPEV only handles a subset of the OCaml type system. To extend the current OPEV type system into new types, one has to manually add specialized test generating heuristics in OPEV for the new types.

4.3 Test Generation

Types in the `PBasic` and `PComplex` classes have corresponding built-in types in OCaml and PVS. Thus, the test generating rules are simple and straightforward. OPEV generates multiple values for every function argument according to its type and then denotes the values to fit them into OCaml and PVS formats.

For example, for the `int` type, OPEV randomly generates an integer in a predefined range ([-10, 10] by default). The integer follows a uniform distribution, and the predefined range can be modified by the user. For instance, for the range [-5, 5], the corresponding command is as follows:

```
./opev --range -5 5 library_path
```

For types in the `PDef`, `PExt`, and `PSpec` classes, we develop more intricate and complex rules to generate the test cases. For example, OPEV only generates test cases for concrete types. Thus, for an arbitrary type, we define a rule that each arbitrary type must be instantiated to `bool` or `nat`, following the built-in test rules in the Lem source code.

4.3.1 Complex Data Types

For complex data types such as `list` and `string`, we set a length parameter that constrains the maximum length of the type element:

```
./opev --length 16 library_path
```

Since these complex data types have corresponding built-in definitions in OCaml and PVS, we do not need to consider the termination problem for some recursively defined data types because we design specific rules for each of these data types.

For example, if the argument type is `list`, OPEV first randomly generates an integer which is the length of the list, constrained by the predefined maximum length parameter. Then OPEV generates elements for the list, following the rules for the list type. The test value for the list is constructed for OCaml and PVS, respectively, following their list representations. For instance, for a list of length `n`, if the list elements are x_0, x_1, ..., and x_{n-1}, OPEV composes an OCaml list as $[x_0; x_1; ...; x_{n-1}]$ and a PVS list as $(: x_0, x_1, ..., x_{n-1} :)$.

4.3.2 User-Defined Types

In OCaml, developers can apply the `type` keyword to define a new type that represents a `record` or a `datatype`. The newly defined type may have various fields, and each field is denoted with a specific constructor and the corresponding type annotation. OPEV sequentially constructs test-cases for each field of the user-defined type. However, this may cause an infinite loop when there are recursive definitions in the user-defined type; thus, we set a maximum limit of recursive times to prevent infinite construction. Additionally, if the return type is a user-defined type, OPEV requires additional construction rules to directly translate the return results from OCaml to PVS, which means that, if a developer intends

to use OPEV to generate tests for a new user-defined type, he/she needs to implement the construction function in the source code of OPEV.

4.3.3 External Types

To automatically generate test cases for the case studies (Chapter 5), we define generation rules for some external types that are used in these libraries. External types are the OCaml types imported from external libraries, which means we do not know the detailed implementations of the interfaces regarding these types. We have to manually design specific mapping functions from the OPEV intermediate type to OCaml external types and PVS types.

For instance, in our case studies, a typical external type is `Nat_big_num.num`, which is introduced in the library file `nums.cma`. This type is employed to handle the situation where there are large integer operations. However, in PVS, there are no limitations on the range of the default `int` and `nat` types. Thus, in PVS, the test cases can be generated following the rules for `int` and `nat`. On the other hand, in OCaml, we introduce a mapping function named `Nat_big_num.of_int`, which converts an integer into a `Nat_big_num.num` number.

4.3.4 Functional Types

The challenge of constructing a functional argument lies in that the function domain and range are potentially infinite. We initially considered applying the methods in Haskell QuickCheck [25] to generate a functional argument; however, the generated function might have different behaviors in OCaml and PVS because they take random generation seeds. Since we have to generate equivalent functions for OCaml and PVS, we designed a comparatively simple method to generate the functional argument.

First, we define multiple functions in PVS with some specific function patterns. Then OPEV randomly selects a predefined function and applies the function name as the PVS argument. Meanwhile, the OCaml argument is the corresponding function name related to the PVS one.

However, if there are no predefined PVS functions for certain patterns or there are no matching OCaml functions, OPEV constructs a `LAMBDA` expression to take symbolic arguments as the inputs and return a randomly generated constant as the output. This `LAMBDA` expression directly serves as the PVS argument, and a corresponding `fun` expression is built as the OCaml argument.

4.3.5 Dependent Types

The generation tactic for a dependent type is to construct the arguments according to its supertype, complying with the constraints of the dependent type. Right now, the supported constraints include arithmetic and comparison operations. Aside from these types of constraints, OPEV will directly generate test cases according to the supertype.

For example, a dependent type in PVS named `word` is defined as follows. `word` is a subtype of `nat`, and the `word` type is constrained by the constant `N`. OPEV uses the constraint to set up a new range for the natural number and generate a natural number within the range as a `word` type argument.

```
word : TYPE = {i: nat | i < exp2(N)}
```

This test construction strategy does not support more complicated constraints than arithmetic and comparison operations, as those would result in some redundant test lemmas that OPEV would reject. Although such test lemmas do not cause any inconsistency for the

OCaml and PVS equivalence, they narrow the test coverage for functions with arguments of these dependent types.

4.4 Proof Automation

For each PVS function, OPEV can automatically generate thousands of test lemmas. It is impractical to manually prove all of them. To automate the proof process, we prove 392 general theorems that support fundamental properties of many translated functions, such as the commutativity and associativity of add operations for bit-vectors with the same length, Listing 4.4.

Listing 4.4: A general PVS theorem.

```
minus_eq_plus_neg: LEMMA FORALL (n:nat, m:nat, bv1:bvec[n], bv2:bvec[m]): m = n
    IMPLIES bv1 - bv2 = bv1 + add_vec_range[m]((bv2), 1)
```

Then we implement generic PVS strategies using these general theorems according to the patterns of the functions that are being tested.

For example, in Listing 4.4, the theorem named `minus_eq_plus_neg` proved that the subtraction of two bit-vectors is equivalent to the addition of the first bit-vector and the negation of the second bit-vector. With this theorem, testing regarding bit-vector subtraction operation can be rewritten to addition operation and negation operation.

With the pre-implemented PVS strategies, we then leverage a utility in PVS called Proof-Lite [58] to prove the test lemmas on these functions. The strategies will be able to instantiate these general theorems with concrete numbers as need be in the test lemmas. Moreover, Proof-Lite verifies the test lemmas sequentially. Therefore, we design a `memory management`

`algorithm` to prove the test lemmas concurrently while efficiently utilizing memory. In the memory management algorithm, OPEV calls multiple processes to verify the test lemmas concurrently, monitors the status of the running machine, and automatically adjusts the number of activated processes according to the memory usage of the machine.

4.4.1 Automatic Proof Strategies

To automatically prove large-scale test lemmas with non-executable semantics in PVS, we implement a set of generic PVS strategies. To construct a generic PVS strategy for different functions, we start from a single test lemma and prove it manually. During the manual proof procedure, we extract a simple PVS strategy for this test lemma pattern. Then we try to prove other tests with different patterns using this PVS strategy. If this strategy does not work, we manually prove the new tests and get new PVS strategies. Then we try to combine the PVS strategies for different test patterns together using branching, backtracking, or feature extracting and summarizing. By repeatedly carrying out this process, we synthesize the unified pattern behind the verification of the test lemmas. We then construct a generic PVS strategy using the unified pattern. (It is possible to automate this proof generation, possibly using SMT solvers; we scope that out as future work.)

For instance, in the basic OCaml-to-PVS translation (Section 5.1) library, functions mainly focus on bit-vector operations. The functions in this library involve conversions between natural numbers and their corresponding bit-vector representations. This conversion from natural number to bit-vector in PVS is defined as follows (the source code is in [10]):

```
nat2bv(val: below(exp2(N))): {bv: bvec[N] | bv2nat(bv) = val}
```

The `nat2bv` function is non-executable since it just declares that it is the inverse function of `bv2nat`, which defines the conversion from bit-vector to natural number. Meanwhile, most

of the functions in the OPEV_Value library call this `nat2bv` function. Thus, we can exploit the relation between `nat2bv` and `bv2nat` to circumvent the execution of `nat2bv` function, which is non-executable, and to prove test lemmas containing `nat2bv` function.

For example, the `case-split-strat` strategy, as illustrated in Listing 4.5, applies the injectivity and invariance properties of the `nat2bv` and `bv2nat` functions. This PVS strategy can be grandly applied to test lemmas for functions in the OPEV_Value (Section 5.1) library.

Listing 4.5: A generic PVS strategy.

```
(defstep case-split-strat (fname &optional (fnum 1))
  (let ((rewritestr1 (format nil "~a_inj" fname))
        (rewritestr2 (format nil "~a_inv" fname)))
    (branch (case-insert-fname fname fnum)
            ((then (rewrite rewritestr1)(grind)(eval-formula))
             (then (hide 2)(rewrite rewritestr2)(grind)(eval-formula))
             (then (grind)(eval-formula)))))
  "" "")
```

After implementing the generic strategy, we apply Proof-Lite, augmented with our memory management algorithm, and the PVS strategy to prove all the test lemmas generated for the functions in the library. We are able to efficiently prove hundreds of thousands of test lemmas automatically. The statistics are illustrated in Chapter 5.

Chapter 5

Case Studies of OPEV

We now illustrate the application of OPEV on two case studies: a manually implemented OCaml-to-PVS translation in Section 5.1 and a Sail-to-PVS parser in Section 5.2. We detected 11 mismatches during the validation of these case studies. Documentation on these errors is available in [9]. The verification was carried out on an AMD Opteron server (2.3GHz, 64 core, 128GB).

5.1 Manually Implemented OCaml-to-PVS Translation

OPEV validated a manually implemented PVS library for which the source is a single OCaml file in the Sail source code [11], which supplies Sail with definitions and operations of bits and bit-vectors. Since the translation is done manually, the translated PVS library is error-prone. It is desirable to increase the reliability of the translation. Table 5.1 illustrates the statistics for this validation.

We verified ∼200K test lemmas and found 6 mismatches. An example mismatch: in the implementation of `add_overflow_vec_bit_signed` function in PVS, if the second operand is false, we then assume that there is no overflow and no carry bit for the addition operation. However, in one version of sail_values.ml [11] (commit ce962ff), overflow is set to true. Thus, there is a conflict in the two implementations and the results parsed from the execution of the OCaml function cannot be verified in the PVS test lemmas. OPEV detected this difference

in intention as an error.

Table 5.1: Statistics on validating the OCaml-to-PVS translation.

OCaml Source Code Size	1,488 LOC
PVS Destination Code Size	1,533 LOC
# of Validated Functions	150
# of Manually Proved Generic Lemmas	268
# of Auto-Generated Test Lemmas	215,562
# of Missmatches Found	6

5.2 Sail-to-PVS Parser

The Sail language [36], which is a first-order imperative language, has been used to describe the semantics of ISAs such as x86, ARM, RISC-V, and PowerPC [36]. To facilitate the reasoning on these semantics, we implemented a Sail-to-PVS Parser to expose the semantics of many ISAs and their multitudes of variants – already available in Sail – to the community of PVS users.

The architecture of the parser is shown in Figure 5.1. First, we rely on the Sail compiler [11] to automatically translate Sail source code to Lem [6], which was designed to serve as a semantic model that was mathematically rigorous [57] and can be translated to OCaml for emulation of testing as well as to Isabelle/HOL, Coq, HOL4, and other languages. Then we employ the Lem compiler to translate the resulting Lem source code into a typed Abstract Syntax Tree (AST). Both the Sail and Lem compilers are in our trusted computing base. (We argue that trusting these two compilers is reasonable due to their small codebase. Besides, they have undergone intensive unit testing in prior work [6].)

Our Sail-to-PVS parser takes this typed AST as input and implements two independent parts: an embedded translator and a rewrite handler. The translator is embedded in the

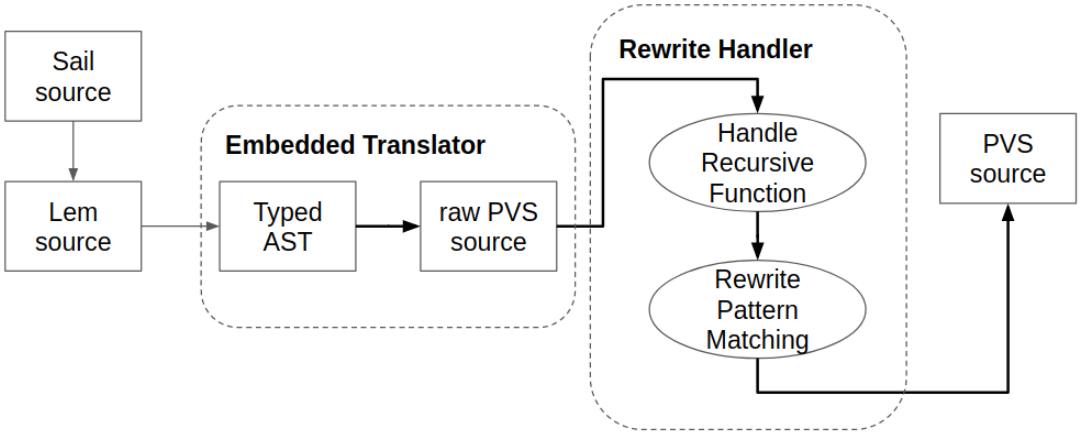

Figure 5.1: Architecture of Sail-to-PVS parser.

Lem source and translates the typed AST into corresponding PVS code using PVS syntax. The Lem type system does not support dependant types and originally was designed to translate Sail specifications into theorem provers that do not support dependant types, such as HOL4 and Isabelle [6]. In addition to this challenge, at this stage, the generated PVS code is challenging and error-prone due to other differences between PVS and Lem specification languages. For example, the method of reasoning about the termination of recursive functions and various formats of pattern matching for different pattern types. To solve the problems, we apply a rewrite handler, written in Python, to adjust the problematic PVS code. The rewrite handler performs two tasks: rewrite the pattern matching to ensure that the PVS code has consistent types and add `measure` functions for all the recursive functions. The total LOC of the Sail-to-PVS parser, including the embedded translator (1,730 lines of OCaml code) and the rewrite handler (1,033 lines of Python code), is 2,763. However, with these modifications Sail-to-PVS parser is still restricted to pure functions of Sail.

An important use case of the Sail-to-PVS parser is program verification at the assembly level (using PVS). For such a use case, it is critically important that the translation is provably correct. We automatically translate a Lem basic library [6] respectively to PVS

and OCaml using the Sail-to-PVS parser and Sail's built-in compiler. Although Sail and Lem are executable, the generated PVS code would call some built-in PVS functions, some of which are non-executable; however, all of them are pure. Since the generated OCaml code is within the scope of OPEV's OCaml subset, it enables us to validate the equivalence between the generated OCaml and PVS code using OPEV. If the equivalence is validated, our trust in that the Sail-to-PVS parser carries out similar functionality as the Sail built-in compiler will increase significantly. Thus, Sail-to-PVS parser is reliable if the Sail built-in compiler is trustworthy.

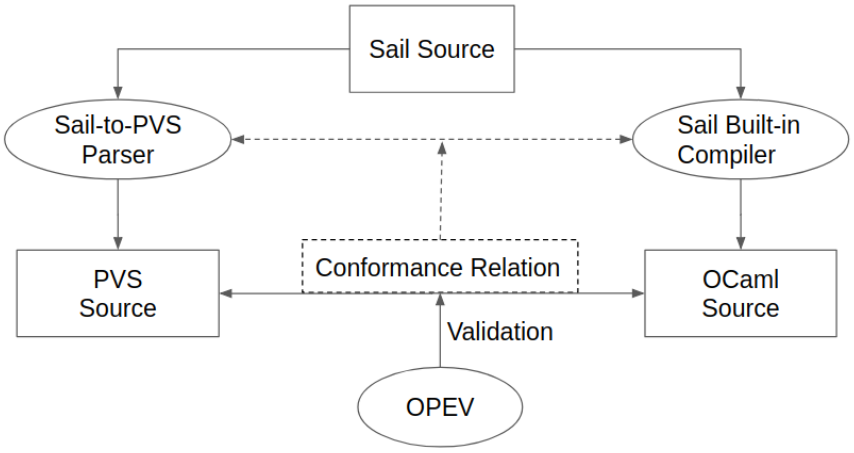

Figure 5.2: Application of the OPEV methodology to validate the Sail-to-PVS parser.

We generated small-scale test cases at the beginning, namely 10 test cases for each function, and attempted to prove all the test lemmas by a default PVS strategy called **grind**. For the test lemmas that cannot be proved, we designed the PVS strategies by proving auxiliary lemmas or by combining multiple strategies together according to the steps described in Section 4.4. Then we generated large-scale test lemmas and verified them using the corresponding strategies.

Table 5.2 shows the statistics for the library. OPEV determined multiple unprovable test lemmas in the PVS implementation. In turn, we modified the source code of the Sail-to-

Table 5.2: Statistics on validation of Sail-to-PVS parser.

Lem Source Code Size	7,542 LOC
PVS Destination Code Size	10,990 LOC
# of Validated Functions	109
# of Manually Proved Generic Lemmas	124
# of Auto-Generated Test Lemmas	242,685
# of Missmatches Found	5

PVS parser, which generated the test lemmas reported in the table. Due to the gap between the semantics of the Lem and PVS languages, OPEV detected 5 mismatches. Doing this translation validation is practically impossible to achieve manually without OPEV.

Chapter 6

DSV: Disassembly Soundness Validation

To evaluate whether a binary file is correctly disassembled requires a lot of sophisticated work. For instance, some inline data, such as jump table, is possible to be embedded in the code section. It is undecidable to distinguish instructions from raw data. Moreover, predicting where indirect branches jump to is a major challenge that almost all the disassemblers are committed to finding solutions.

The evaluation is more challenging when there is no source code for the binary. Since programming languages, whether imperative, object-oriented, or assembly, have specific semantics and are human-readable, researchers can construct the model of these languages and verify the soundness on these models. However, machine instructions are written in a binary file with binary code. Thus, the formal validation of the soundness by verifying model consistency is infeasible here. Validating the soundness of disassembly by testing is a feasible method. However, it is difficult to monitor the running result for every single instruction in the binary execution. Besides, the reliability of testing is low since it cannot cover all the possible paths during the execution.

In this chapter, we describe a definition of soundness of a disassembly process in Section 6.1. We extend the definition to fit for more realistic cases in Section 6.2 by taking into account different instruction formats implemented by different disassemblers. Section 6.3 demon-

strates the false positive and false negative of our definition.

6.1 Definition of Soundness

To formulate a formal notion of soundness of a disassembly process, we first provide definitions of the concepts used in that formulation. We use the terminology N*word* to refer to the bit sequence with length N in a binary file. |N*word*| represents the length of the bit sequence, which is N.

The first main function **ReadWords** reads multiple words from a binary file using the starting address and the size of the words.

$$ReadWords : 64word \rightarrow \mathbb{N} \rightarrow [8word]$$

To express the symbolic execution of the instructions in the assembly file, we use the $\rightarrow_A$ function to denote the execution step from one instruction address to the set of next instruction addresses. Some conditional jump instructions provide multiple next addresses. However, if the current state of the computer system is determined, then every instruction has one next address. We use $\rightarrow_A^*$ to indicate the transitive closure of the execution, which means, $\rightarrow_A^*$ maps one address to another address by repeated execution.

$$\rightarrow_A : 64word \rightarrow 64word$$

Another function, **bytes**, maps an instruction to its corresponding words expression. This is the basic work of most assemblers. For example, $bytes(xor\ ebp, ebp) = [31, ed]$.

$$bytes : Instruction \rightarrow [8word]$$

Finally, the output of a disassembler is represented as **D**, which is a mapping from an address to the corresponding instruction in the generated assembly file. We use $64word$ since addresses are represented using 64 bits in 64-bit mode.

$$D : 64word \rightarrow Instruction$$

We collectively write the instruction address in the assembly file as a. The entry address in the binary file is denoted as *entry*, and each single instruction inside the assembly file is represented as I.

Definition 6.1. The output of a disassembler D is *sound*, if and only if:

$$\forall a.\, entry \rightarrow_A^* a \implies bytes(I) = ReadWords(a,\ |bytes(I)|)\ where\ I = D(a) \tag{6.1}$$

Definition 6.1 means that for all addresses a inside a binary file, if an address a is reachable from the entry address *entry* by repeated symbolic execution. Then we read the corresponding instruction I from the assembly file using its address a. The byte-representation of the instruction I (by **bytes** function) should be equivalent to the words that are directly read from the binary file using the address a and the size of the words.

This definition is independent of what algorithm has been applied by the disassembler. Whether a disassembler is implemented using recursive traversal, linear sweep or machine-learning is irrelevant since we are trying to prove the behavioral consistency between a binary file and the corresponding disassembled assembly file. We treat the disassembler as

a black-box and only consider the output.

6.2 Extending the Soundness Definition

In the previous section, we provide the soundness definition of disassembly. In this section, we give explanations to the soundness definition. Our definition works well on the ideal cases, where the binary file that is generated from the assembly file is strictly equivalent to the original binary file. However, since many disassemblers employ different levels of optimizations during the disassembly process, the generated binary file is possibly different from the original binary. For these special cases, we extend our definition to apply to them.

6.2.1 Ideal Cases

To this end, we define the assembly process as the **asm** function and disassembly as the **disasm** function. Suppose the original assemble code is asm_0, then we get

$$asm(asm_0) = bin_0 \tag{6.2}$$

Here bin_0 is the generated binary file which is the source of our soundness definition. Note that asm_0 is not introduced in our algorithm, we just use it to illustrate the validity of the algorithm. Then we apply the **disasm** function to bin_0 and get the following formula

$$disasm(bin_0) = asm_1 \tag{6.3}$$

In Definition 6.3, asm_1 is the other end of our verification. Our definition is to verify the soundness of the **disasm** formula. Since we only have bin_0 and asm_1, we apply the **asm**

function to asm_1, and get bin_1.

$$asm(asm_1) = bin_1 \tag{6.4}$$

At this point, we can compare bin_0 and bin_1. If they are equal, we can infer that asm_0 and asm_1 are equal according to Equation 6.5 (There is an implicit premise that the **asm** function should be injective, which holds true for most assemblers). Then we get the conclusion that the **disasm** function is sound since the generated asm_1 and the original asm_0 have the same initial design intent, although we do not have the source code of asm_0.

$$bin_0 = bin_1 \Longrightarrow asm(asm_0) = asm(asm_1) \Longrightarrow asm_0 = asm_1 \tag{6.5}$$

This is the ideal situation that we can check the soundness of most disassembly process with the Definition 6.1.

6.2.2 Special Cases

Since the assemblers and disassemblers in action would take different optimizations during the assembly and disassembly procedures, the disassembly result sometimes does not meet the soundness definition. For example, we employ gcc as the assembler and objdump as the disassembler and get the example in Listing 6.1.

Listing 6.1: An example that does not satisfy the soundness definition.

```
objdump(0f 1f 44 00 00)=nop DWORD PTR [rax+rax*1+0x0]
gcc(nop DWORD PTR [rax+rax*1+0x0])=0f 1f 04 00
objdump(0f 1f 04 00)=nop DWORD PTR [rax+rax*1]
```

In this example, bin_0 is `0f 1f 44 00 00`, bin_1 is `0f 1f 04 00`, and they are not equivalent to

each other. If we simply compare bin_0 and bin_1, we would make the wrong declaration that the disassembly process carried out by objdump is not sound. However, the disassembled result is sound since `nop DWORD PTR [rax+rax*1+0x0]` and `nop DWORD PTR [rax+rax*1]` are the same results with different representations. The reason behind this situation is that gcc would automatically take optimization when it encounters certain type of instructions (such as nop). Thus, we set special cases for these kind of instructions and extend the scope of the soundness definition. Then we get the following extensions.

$$bin_0 \neq bin_1 \tag{6.6}$$

For some special instructions, the bin_0 and bin_1 are not equivalent to each other. In this situation, we reapply the **disasm** function to bin_1 and get a new asm_2.

$$disasm(bin_1) = asm_2 \tag{6.7}$$

Then we make a comparison between asm_1 and asm_2, if they are instructions with equal op code and same operands, we identify that they represent the same instruction, which are expressed as $asm_1 \sim asm_2$.

$$\begin{aligned} asm_1 \sim asm_2 \Longrightarrow disasm(bin_0) \sim disasm(bin_1) \Longrightarrow \\ disasm(asm(asm_0)) \sim disasm(asm(asm_1)) \Longrightarrow asm_0 \sim asm_1 \end{aligned} \tag{6.8}$$

Then we provide an extended verification of the soundness definition as shown in Equation 6.8. In Equation 6.8, if asm_1 is similar to asm_2 and **disasm** is the exact reverse function of **as**, we conclude that asm_1 is similar to asm_2. Here the similarity means that the assembly codes take the same behaviors. The extended implementations of soundness resolve a large number of special cases.

6.3 False Positives and False Negatives

In Section 6.1, we define the soundness of the disassembly process. We extend the soundness definition in Section 6.2 to make it suited for the real cases that are generated by different disassemblers. According to the soundness definition, we implement a tool called DSV, which takes a binary file and a generated assembly file as input, to check whether the disassembly process is sound or not. It means that the outputs of DSV have two possibilities: sound or unsound. On this basis, we explain the false positive and false negative of our tool in this section.

For a binary file and specific disassembler, **false positive** is the case that the disassembly process is validated as sound, however, there are errors, such as mistakenly disassembled instruction or missed instructions in the disassembly process. As an informal argument, DSV does not generate **false positive** results in the execution since DSV over-approximates the execution paths during the symbolic execution. That is, if the execution of specific instruction is undecidable and the value of operand inside the instruction is symbolic, DSV approximates all the possible paths to check the soundness of the disassembly. Since in the real execution, some of the paths are infeasible, this over-approximation of the path exploration prevents DSV from the generation of **false positive** cases.

On the opposite, **false negative** indicates the situation that the disassembly process is correct; however, it is classified as unsound during the soundness validation. **False negative** situation is possible in the validation process using DSV. The reason also lies in the over-approximation of DSV's implementation. For example, the failure of the soundness validation is caused by some paths during the symbolic execution. However, these paths are infeasible in real execution. The over-approximated paths lead to the **false negative** validation.

As shown in Fig. 6.1, DSV supports **true positive**, which indicates soundness of the disas-

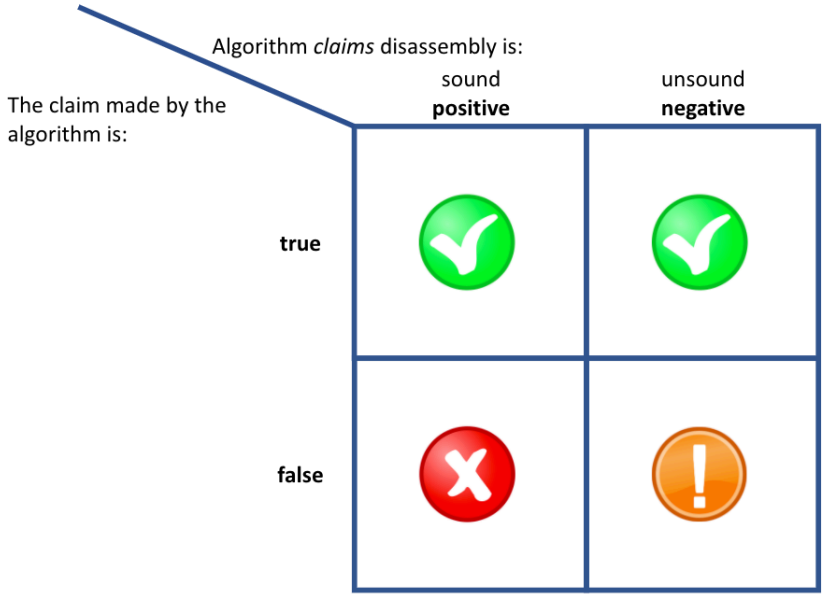

Figure 6.1: False positive and false negative analysis for DSV.

sembly process, and **true negative**, which denotes that the disassembly is unsound. Besides, it is possible that the validation results generated by DSV is **false negative**. As an informal declaration, DSV does not support **false positive** results.

Chapter 7

DSV Implementation

In the soundness definition, we set a premise that we will check the soundness of disassembly between byte sequences and instructions whose addresses are reachable from the entry point. There are two approaches for the implementation. The first one is to simulate the execution in the binary file, which is challenging since it is hard to model the binary file and monitor the status of CPU and memory. Thus we apply the second method, which is to construct a symbolic execution on the assembly file and check the soundness of every reachable instruction during the execution. We implement a tool called DSV to validate the disassembly soundness.

In this chapter, we introduce the major steps for the implementation of DSV in Section 7.1. Then we present the state model of a computer system in Section 7.2. Section 7.3 introduces the model of instruction semantics of X86/64 ISA. We introduce the treatment of external function calls in Section 7.4. Section 7.5 demonstrates the handling of loops during the symbolic execution. Then we introduce the methodologies that we develop to resolve the indirect jump addresses in Section 7.6.

7.1 Major Steps

In Section 6.1, we define four major functions: **ReadWords**, **bytes**, **disasm**, and $\rightarrow_A$. For each of them except for function **disasm**, which is the output of specified disassembler, we

applied different methods to implement the functions.

ReadWords reads byte sequences from a binary file. We employ *readelf* to get the binary section information and implement a Python program to directly read from the binary file.

To implement function **bytes**, which means we need to translate a single instruction to its byte sequence representation, we wrap up the instruction to a temporary file and directly apply *gcc* to generate the byte sequence. Here, *gcc* can be replaced with *llvm* depending on the type of the original assembler.

The function that is most difficult to implement is $\rightarrow_A$, which maps from one address to a set of next addresses during the execution. Since there are multiple issues arise during the symbolic execution, we use the whole Chapter 7 to explain the detailed implementation and our design ideas.

7.2 State Model

To simulate the execution of the assembly file, we build up a tool called DSV which carries out symbolic execution. The reason we use symbolic execution lies in that the status of registers, stacks, and memories is unknown at the beginning of the execution. Besides, some instructions would set the status of a machine to an unknown situation. In such cases, we set the status of these registers or memories to symbolic values without concrete values and continue the execution of the instructions.

In our symbolic execution implementation, we maintain the current state of the execution in each block and maintain the branch predicate in the edges that points from one block to another block. Besides, each block takes a record of the constraints, which is a conjunction of predicates that we take during the branch selection of the symbolic execution.

The general elements of a computer system, whose status is modified during the execution of the binary file, include registers, memories, flags, etc. We build up our state model using registers, memories, and flags since these are the most critical factors that affect the execution of instructions.

7.2.1 Flags

The flags register is simplified to 5 flags which are respectively referred to CF, ZF, OF, SF, and PF. To simplify the operation, we do not use symbolic value to represent these 5 flags. Instead, we apply True/False/None to these flags to indicate the concrete and unknown states of corresponding flags. Although introducing the flags to the state model advances the construction of instruction semantics in Section 7.3, we can prune some infeasible paths with these flags since some conditional jump instructions take branch according to with these flags.

7.2.2 Registers

The number of registers is limited in the computer system, thus we model the general-purpose register as a 64-bit Z3 bit-vector. We introduce the register to the state model when the value of the register is related to certain instructions. And we update the value of the register according to the overwriting rules of the registers.

7.2.3 Memory

There are different methods to model the memory of a computer system. Since our model needs to simulate the execution of a computer system, we intend to design a time- and space-

efficient memory model at the very beginning. The whole memory of a computer system, including the heap and stack, is modeled as a function **mem**, which mapping from memory address to words and its size (the unit is the length of bytes in this memory block).

$$mem : address \longrightarrow (words, size)$$

This mapping is partial, which means that not all addresses have corresponding content in this memory model. Then how to read and write memory content is designed according to the features of this model.

When we read the memory content with given address a and specific size sz, if a does exist in the domain of mem function, then we try to read the specified memory content with a given size. Otherwise, we check the continuous addresses from $a-1$ to $a-7$ to see whether some of the address does exist in our memory model since it is possible that $mem(a-i)\,(i = 1...7)$ does contain the value in $mem(a)$. If so, we try to split the required memory content from $mem(a-i)$; alternatively, we have to return a random symbolic value as the memory content.

To write some value to the memory at address a and size sz, we have to check the continuous addresses from $a-7$ to $a+7$ to see whether the writing operation would affect the content of some of these addresses. If so, we have to split the memory content and replace the original memory content with the updated value. Otherwise, we directly add the new address a and (value, sz) pair to the mem function.

For example, in Fig. 7.1, before the execution of `mov QWORD PTR [1000], 0xaaf1343` instruction, these are two addresses in the domain of function mem: address 998 and 1004. Thus we have $mem(998) = (0x10002, 4)$ and $mem(1004) = (0x0012, 6)$. Now if we need to read the memory at address 1000 with size 2, then we get the result as $0x1$, which is splitted from $mem(998)$.

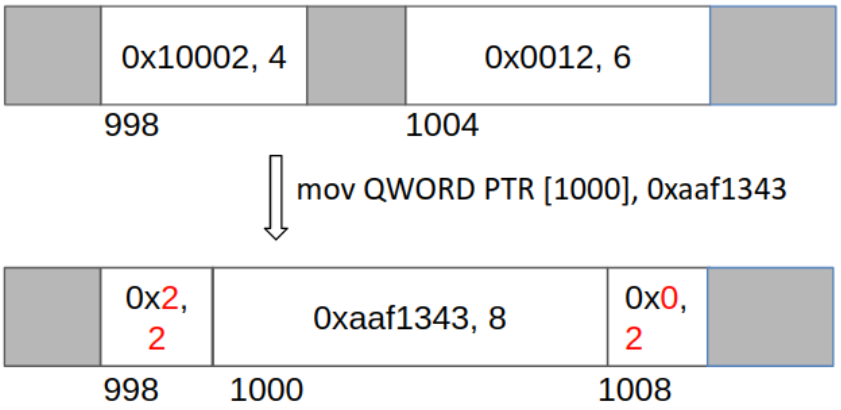

Figure 7.1: A memory model example.

Then after the execution of `mov QWORD PTR [1000], 0xaaf1343` instruction, we have to write 0xaaf1343 with size 8 to address 1000. This will affect the result at both address 998 and 1004. After the overwriting, there are three addresses in the domain of the updated mem: 998, 1000, and 1008. Thus we have $mem(998) = (0x2, 2)$, $mem(1000) = (0xaaf1343, 8)$, and $mem(1008) = (0x0, 2)$.

7.3 Instruction Semantics

After we build up the model of registers, flags, and memory, we need to construct the semantics of x86 instructions to indicate what changes will be made on the state model. It is not necessary to model the semantics for all the instructions since many floating-point related instructions do not affect the status of registers that refer to some indirect jump address. Thus we construct the semantics of 44 basic instructions and instruction sets, and the introduction of the semantics comes from [54]. Besides, we model the semantics of direct/indirect jump and return instructions to construct the control flow graph (CFG) that enables us to figure out the next execution addresses for every reachable address.

7.3.1 Bottom Semantics

There is no need to set up complete semantics for *all* instructions. In our implementation, instruction semantics is constructed to change the value of the RIP register to guide the symbolic execution. Thus, we only need to build up semantics for instructions that influence the RIP register. These instructions include MOV, LEA, simple arithmetic instructions, etc. Advanced instructions such as PUNPCKLBW (for interleaving byte sequences), floating-point instructions, or SIMD extensions typically do not impact the RIP. We do not construct specific semantics for these instructions.

We introduce bottom semantics to our implementation to fill the vacancy made by the unimplemented instructions. In general, an instruction has an opcode and different operands; and the content of the destination operand is modified by the instruction. If the semantics of the instruction is unimplemented, we set corresponding destination operand to a special symbolic value called **bottom**. The **bottom** symbol represents that the current status of the corresponding register, flag, or memory is undefined, or undetermined.

To deal with the operations on **bottom** symbol, we develop an extended instruction semantics. For most instructions, if any of the operands is **bottom**, then the destination operand is also set to **bottom**. However, some instruction, such as *xor ebp, ebp*, the status of *ebp* is set to 0 even if its initial value is **bottom**.

Though the import of **bottom** semantics saves a lot of time and effort in building up the instruction semantics, it also raises certain issues. For instance, what should we do if the jump address of certain branch instruction is **bottom** symbol? In such a situation, we trace back the CFG and re-implement the semantics of the relevant instructions. The details of the trace-back implementation are explained in Section 7.6.1.

7.4 External Function Call

With the introduction of state model and instruction semantics, we construct a control flow graph (CFG) by symbolic execution, which assists us to determine the reachability of each instruction and validate the soundness of a disassembly process. When establishing a CFG, we have two inputs: a binary and an assembly file disassembled from the binary file.

Generally, the binary file has no source assembly code and is compiled non-statically, thus some critical external functions, such as *printf*, *malloc*, and *rand*, are dynamically linked during the execution. On the other hand, when we execute our symbolic execution, the assembly file does not have any detailed information for these external functions since the assembly file is directly disassembled from the binary file. These external functions modify the status of the state model while there is no source code for these functions. Thus we need to implement special methods to handle these external functions when constructing the CFG.

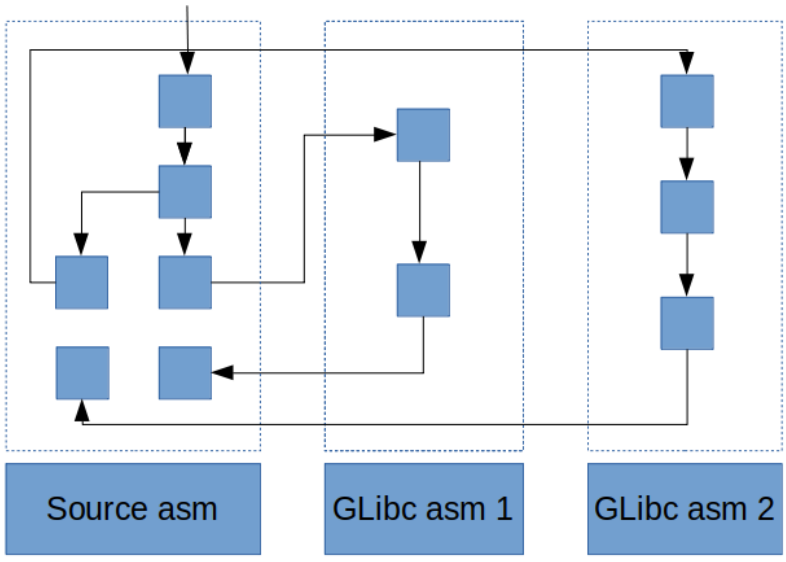

Figure 7.2: Recursive construction for external functions.

Since most external functions are imported from glibc library [3] and glibc is written in

C code, we compile and disassemble glibc to get the assembly code and exploit the generated assembly code as part of our trustbase. As shown in Fig. 7.2, when we encounter an external function during the CFG construction, we locate the function in the glibc assembly library and jump to that function to recursively construct another CFG for the external function. Finally, all the CFGs are inter-connected. There exists an exception, which is *___libc_start_main*, in all the external function calls. This function is excessively complicated and cannot terminate naturally since it starts up a new thread to execute the **main** function while our construction is executed for the single-thread program. For *___libc_start_main* function, we set all the modified variables to **bottom** and then directly call the **main** function to ensure that our construction can terminate automatically.

Our method of dynamically loading glibc library and recursively constructing CFGs does support the commercial or legacy binaries that are compiled non-statically, which enables us to employ our implementation on a wide range of applications. On the other hand, the simulation of dynamic linkage is time-consuming. Besides, we include the assembly code of glibc in our trustbase without any validation, which could become a source of untrustworthy.

7.5 Loop

A substantial but common challenge in CFG construction is the path explosion problem. To handle the problem, we prune some infeasible paths using the model of flags. Besides, in our implementation, DSV reduces memory usage by sharing unchanged states between multiple blocks. However, there still exist the cases that the construction fails due to running out of resources. A major reason for this kind of failure comes from the infinite loops existing in the execution of assembly code.

For a bounded loop, we simply unroll the loop execution and construct the CFG. However,

the infinite loop would cause a path explosion problem that disables the termination of the symbolic execution. To solve the problem, we need to extract the loop invariant during the execution. We have implemented a state-merge algorithm to generate the loop invariant. That is when we execute an instruction the second time and the state s' is different from the state s in the first execution, we locate the changed variables and set these variables to the *bottom* symbol. Then we continue the execution until the newest state, which is the loop invariant, does not change anymore. This method is straightforward. However, the application of a *bottom* symbol aggravates the resolving of indirect jump addresses.

7.6 Indirect Jump

In the construction of CFG, most instructions are executed sequentially, which means we can get the next execution address directly from the assembly file. For jump and call instructions with direct jump addresses, our implementation can straightforwardly construct branches. However, indirect jump, indirect call, and return instructions lead to the problem of how to resolve the indirect jump addresses. We endeavor to implement as many instruction semantics as possible to resolve the indirect jump addresses. We also introduce a trace-back model and a pattern for the jump table without determined upperbound to solve the problem.

7.6.1 Trace-back Model

The CFG is constructed forwardly by symbolic execution. When we encounter an indirect jump address that is represented as **bottom** symbol, we know that it either comes from some undefined instruction semantics or comes from the undetermined state at some CFG

block. We introduce a trace-back model that repeatedly trace back from the current block to its parent block and check and adjust the operations taken at the parent block.

As an example shown in Fig. 7.3, we suppose that the semantics of `lea` instruction has not been implemented yet. The symbolic execution terminates unexpectedly at block 3 since the instruction at block 3 is `jmp rax` where the *rax* is **bottom**. Then we step back from block 3 to its parent block, which is block 2, and check whether the instruction semantics at block 2 is undefined and whether *rax* is the destination operand of the instruction. Since the instruction at block 2 is `mov rax, rbx`, we know that the value of *rax* comes from *rbx*. We continue tracing back to block 1. At block 1, we figure out that the problem comes from the unimplemented semantics of `lea` instruction. We introduce semantics for `lea` instruction and re-construct the whole CFG from block 1.

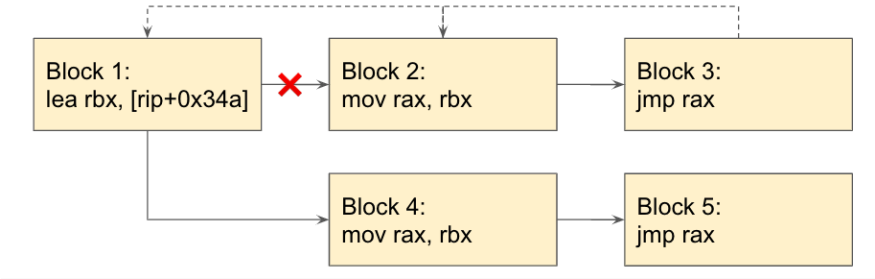

Figure 7.3: A trace-back example.

7.6.2 Jump Table without Upperbound

For source code which includes carefully designed *switch* statement, a compiler generates a jump table at *.data* segment to facilitate the selection between multiple branches. Generally, the address and the index for the jump table are represented with concrete integer values. Thus we can read the content of the jump table directly from the binary file using its starting address and the index. However, if the index of the jump table, which is usually stored in a

register, is represented as **bottom**. Then we cannot construct the CFG correctly since the index of the jump table is not determined and our execution may jump out of the range of the jump table without determined upperbound.

We figure out a specific pattern for this kind of problem and solve the problem with concrete upperbound extracted from the pattern. For example, as illustrated in Listing 7.1, if the value of *rdi* is **bottom** at the beginning of the execution, then we cannot resolve the value of *rax* at the `jmp rax` instruction. Thus, we trace back and find the `cmp rdi, 4` instruction. We figure out that if the value of *rdi* is greater than or equal to 4, then the instruction `ja 799` is taken and the branch goes to address 799. Otherwise, to approximate the execution of each branch, we respectively set *rdi* to 0, 1, 2, and 3, which means that we assign an upperbound to the corresponding jump table as 4. Then we carry out the symbolic execution using different values of *rdi*. Although this method solves a certain pattern of indirect jump issues, it has many restrictions. We will take further research on the pattern and improve the design in future work.

Listing 7.1: A typical pattern for jump table without concrete upperbound.

```
705: cmp rdi, 4
709: ja 799
70b: lea rax, [rip+0x20090e]
712: mov rax, QWORD PTR [rdi+rax*1]
716: jmp rax
```

Chapter 8

Case Studies of DSV

After we implement DSV that validates the soundness definition of the disassembly process, we test the tool and employ the tool on the micro-benchmarks introduced in Section 8.1. We carefully design 7 micro-benchmarks, which are inspired by GNU Coreutils, to show some common situations that appear in real projects. We use gcc as the testing assembler and objdump as the disassembler.

8.1 Micro Benchmarks

In this section, we illustrate the 7 different test cases and apply them to test our tool. Since we have introduced different technical points that we have implemented to solve various challenges, we elaborately design the micro-benchmarks to reflect all these challenges as shown in Table 8.1. Besides, although all the test cases call external functions from glibc, which increases the burden of constructing the CFG, each generated CFG contains less than 1,000 blocks. That enables us to directly check the results. Thus, the testing results for all the micro-benchmarks listed in Table 8.1 are checked manually.

In *objects* program, a *struct* data structure is applied to call functions. The assembly code for this program is simple and straightforward. Although this program calls 4 glibc functions, including *free*, *puts*, *printf*, and *malloc*, the internal structure of these four functions are easy to traverse over. Finally, we generate a CFG with 128 blocks, check the reachability of all

Table 8.1: Analysis on various micro-benchmarks.

Microbench	**Features**	**Assembly Size**	**# of Blocks**	**Result**
objects	function call	336 LoC	128	Sound
goto	external function call	308 LoC	465	Sound
callback	callback function	321 LoC	629	Sound
jump_table	jump table	227 LoC	367	Sound
function_pointer	command line arguments	316 LoC	188	Sound
switch_input	bounded loop	294 LoC	273	Sound
indirect	jump table w/o determined upperbound	276 LoC	247	Sound

the addresses, and validate the soundness of the translation.

In *goto* program, it calls the *time*, *srand*, and *rand* functions from the glibc library. Our tool handles these functions and generates a CFG with 465 blocks. It is worth mentioning that the completeness of the CFG branches bothers us at the beginning since *rand* function is a pseudo-random function in the real implementation. Due to the assignment to some internal state in *srand* and *rand* function, *rand* function may generate the similar results in different rounds. However, based on our testing results, *rand* function would cover all the branches after multiple rounds of execution.

Test case *callback* is similar to *goto* in that it also applies *time*, *srand*, and *rand* functions to decide the execution branch. A brand-new feature that we introduce to *callback* is the usage of $argc$ argument of the main function. Although the value of $argc$ is undetermined at the beginning of the simulation, our tool takes care of the undecided $argc$ and constructs a CFG with 629 blocks.

For *switch* statement, some compilers optimize the generated binary by translating a simple jump table at the .data segment to inline code at the .code segment. In order to test our

tool on the jump table, we build up a test case *jump_table* to generate the jump table in the .data segment using a simplified example from [1]. In this *jump_table* example, we simply test our tool's functionality of handling the jump table. Most variables in the example are concrete and the loop round is limited. As shown in Fig. 8.1, the jump table is located at the absolute address 0x201020 and is loaded to register *rax*. Note that the absolute address is the corresponding address when the binary is dynamically loaded to the memory. This address has to be re-calculated by subtracting the offset of the .data segment to get the real address at the binary file. Then the memory content stored at the address with index offset is loaded to *rax*, which is the next jump address. For this test case, we construct a CFG with 367 blocks. The success of this test case validates the tool's functionality of reading the jump table directly from the binary file.

```
70b:   lea    rax,[rip+0x20090e]
712:   mov    rax,QWORD PTR [rdx+rax*1]
716:   call   rax
```

Figure 8.1: Samples from jump_table test case.

As shown in Table 8.1, program *function_pointer* exploits *argc* which is determined by command-line arguments. Besides, it calls some rather complicated function from the C **math** library. Due to the design of the program and our tool, we build up a CFG with 188 blocks from this program.

For the jump table without determined upperbound, we elaborately design a test case called *indirect*. As shown in Fig. 8.2, this test case has the pattern that we have mentioned in Section 7.6.2. Thus, we employ our tool on this test case and construct a CFG with 247 blocks.

We test our tool and validate the soundness of the disassembly process by the above-mentioned test cases. The validating results are relatively reliable since we repeatedly check

```
71a:   cmp    DWORD PTR [rbp-0x14],0x2a
71e:   ja     799 <main+0x8e>
720:   mov    eax,DWORD PTR [rbp-0x14]
723:   lea    rdx,[rax*4+0x0]
```

Figure 8.2: Samples from indirect test case.

all the generated CFGs line by line for the micro-benchmarks. On the other hand, we expect that our tool can detect unsoundness during the disassembly process. For this purpose, we manually modify some generated instructions, either the opcodes or the operands, to check whether our tool can detect these errors. All the man-made errors regarding the correctness of instructions are disclosed by our tool.

8.2 Discussion on Limitations

Although we have defined an ideal soundness definition for the disassembly process, the real implementation has many limitations. First, we have not figured out any perfect solution for the infinite loop problem. The path explosion caused by infinite loops blocks us from constructing a complete CFG and validating the soundness, which disables us from validating some real libraries such as GNU Coreutils. Second, since we have not modeled sufficient instruction semantics, there are cases that DSV cannot resolve the indirect jump addresses for some real applications. The whole x86/64 ISA needs to be further analyzed. Another limitation is glibc assembly library, which is included in our current trust base. The glibc assembly is disassembled from the glibc binary file and we have not checked the soundness of the glibc assembly code, which could be a source of unsound disassembly.

Chapter 9

Conclusions

Translation validation is an important link at many abstraction layers in computer systems which calls for the high reliability of the translation. Each translation validation technique has its own advantages and disadvantages. Empirical testing has short development cycles, however, it is incapable for exploring the entire state space to find all the bugs. Meanwhile, formally verified translators and refinement proofs provide higher levels of reliability. However, they incur significant person effort to develop the formal model for both the languages and to build the conformance and refinement proofs for the two models. There are no uniform or omnipotent methods to solve this problem. A validation method is often selected based on the desired degree of reliability, development time, costs, and features of different languages.

In this dissertation, we present two methods to validate the translation between various languages. The first method is a combination of testing and semi-automatic proofs for languages which have non-executable semantics. The second method is symbolic execution built upon a formal definition of the soundness of the translation between the source and destination languages.

First, we presented a translation validation methodology, called OPEV, that provides high reliability on the translation between OCaml and PVS specifications. OPEV employs an intermediate type system to capture the commonality of the subset of OCaml and PVS and generates test cases for both OCaml and PVS implementations. The reliability of the

translation is ensured by executing large-scale stress tests and automatically proving test lemmas using generic PVS strategies. We demonstrated the OPEV methodology on two case studies, namely, a manual OCaml-to-PVS translation and a Sail-to-PVS parser. OPEV generated more than three hundred thousand test cases and proofs for these case studies and detected eleven errors. OPEV significantly increases our trust in the translations.

The dissertation's second contribution is the DSV methodology, which proposes a formal definition for the soundness of disassembly process. This definition can be used to verify the soundness of disassemblers when source code is not available. We implement a tool, also called DSV, that takes as input, raw binary and assembly code that is disassembled from the binary using an off-the-shelf disassembler, and verifies that each disassembled instruction is correct and reachable from the binary's entry point. To illustrate DSV, we use seven micro-benchmarks, inspired by the GNU Coreutils library. DSV is shown to verify the soundness of these micro-benchmarks.

9.1 Proposed Post-Preliminary Work

We propose three post-preliminary research directions. These are described as follows.

9.1.1 Indirect Branching

In disassembly process, how to resolve an indirect branching address is a major challenge that researchers need to solve. As the analysis in [14] demonstrates, there exists unresolved indirect jump addresses on highly optimized binaries for state-of-the-art disassemblers that have best performance such as IDA pro.

In this dissertation, we have incorporated many techniques to solve the indirect branching

challenge, such as applying a trace-back model and introducing pattern-matching to resolve the jump-table without specific upperbounds. In post preliminary exam work, we intend to model a more complete and trustworthy instruction semantics to solve this problem.

9.1.2 Enhanced Infinite Loop Algorithm

Our current implementation experience suggests that the DSV tool works well on loops that have bounded number of iterations. However, the path explosion problem happens when there exist loops whose bounds cannot be statically determined. This is a major challenge that we propose to solve in our post preliminary exam work.

We have sketched some alternative algorithms, such as merging the states during the execution to generate the loop invariant, or applying abstract interpretation to determine the loop invariant. We propose to fully develop these algorithms, apply them on large-scale benchmarks, and characterize their efficiency and accuracy.

9.1.3 Comparison on Various Disassemblers

We evaluated the DSV tool on micro-benchmarks that were disassembled using objdump, which is a typical linear disassembler. The results on the micro-benchmarks help us to obtain a preliminary understanding of DSV's effectiveness since we do not deliberately set up cases such as overlapped instructions in the test cases. For example, since different disassemblers support different ISA formats, we have to normalize the generated assembly code to the same format and then validate the soundness of the disassembly. How to design a reliable normalization algorithm is one challenge. Besides, to test the false positive and false negative rates, we have to elaborately design test cases that do not influence any specific disassembler, irrespective of whether it is linear or recursive.

9.1.4 Application on GNU Coreutils

Finally, we propose to apply the DSV tool to a complex test suite, such as GNU Coreutils, to better understand the coverage and the performance of the methodology and the tool. We also propose to apply the tool on the glibc library to validate the soundness of the generated assembly code. In the current version, we include the glibc library as part of our trustbase. Formal validation of the library would reduce our trustbase and thereby increase the reliability of the soundness validation of the disassembly.

Bibliography

[1] Branch table. https://en.wikipedia.org/wiki/Branch_table. Last accessed: 2020-03-08.

[2] Ghidra software reverse engineering framework. https://github.com/NationalSecurityAgency/ghidra.

[3] The gnu c library (glibc). https://www.gnu.org/software/libc/. Last accessed: 2020-03-07.

[4] Gnu coreutils. https://www.gnu.org/software/coreutils/manual/coreutils.html.

[5] Ida pro. https://www.hex-rays.com/products/ida/.

[6] Lem project. https://github.com/rems-project/lem. Last accessed: 2019-05-31.

[7]

[8] Ollydbg 64 version 2.04. http://www.ollydbg.de/. Last accessed: 2020-03-07.

[9] OPEV bug report. https://github.com/ssrg-vt/OPEV/blob/master/BugReport.pdf.

[10] PVS source code. http://www.csl.sri.com/users/owre/drop/pvs-snapshots/.

[11] Sail project. https://github.com/rems-project/sail. Last accessed: 2019-05-31.

[12] The sel4 microkernel. https://github.com/seL4/seL4.

[13] Romain Aïssat, Frédéric Voisin, and Burkhart Wolff. Infeasible paths elimination by symbolic execution techniques. In *International Conference on Interactive Theorem Proving*, pages 36–51. Springer, 2016.

[14] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016.

[15] ARM ARM. Architecture reference manual. armv7-a and armv7-r edition. *ARM DDI C*, 406, 2012.

[16] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.

[17] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. The coq proof assistant reference manual. *INRIA, version*, 6(11), 1999.

[18] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16, 2011.

[19] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 131–146, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14052-5.

[20] David Brash. The arm architecture version 6 (armv6). *http://www. arm. com/pdfs/V6_whitepaper_A01. pdf*, 2002.

[21] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[22] Gianfranco Ciardo and Andrew S Miner. Smart: The stochastic model checking analyzer for reliability and timing. In *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pages 338–339. IEEE, 2004.

[23] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[24] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the number and nature of faults found by random testing. *Softw. Test., Verif. Reliab.*, 21(1):3–28, 2011. doi: 10.1002/stvr.415. URL https://doi.org/10.1002/stvr.415.

[25] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351266. URL http://doi.acm.org/10.1145/351240.351266.

[26] Tool Interface Standards Committee et al. Executable and linkable format (elf). *Specification, Unix System Laboratories*, 1(1):1–20, 2001.

[27] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating pvs specications. 03 2019.

[28] Ruth E Davis. Logic programming and prolog: a tutorial. *IEEE Software*, (5):53–62, 1985.

[29] Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *IEEE transactions on Software Engineering*, (4):438–444, 1984.

[30] Michael J Eager et al. Introduction to the dwarf debugging format. *Group*, 2007.

[31] Anthony C. J. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, pages 25–40, 2003. doi: 10.1007/10930755_2. URL https://doi.org/10.1007/10930755_2.

[32] Anthony C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, pages 187–202, 2015. doi: 10.1007/978-3-319-22102-1_12. URL https://doi.org/10.1007/978-3-319-22102-1_12.

[33] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 243–258, 2010. doi: 10.1007/978-3-642-14052-5_18. URL https://doi.org/10.1007/978-3-642-14052-5_18.

[34] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu I Siminiceanu. Model-checking the linux virtual file system. In *Verification, Model Checking, and Abstract Interpretation*, pages 74–88. Springer, 2009.

[35] Michael JC Gordon et al. The hol system description. *Cambridge Research Centre, SRI International, Suite*, 23, 1989.

[36] Kathryn E Gray, Peter Sewell, Christopher Pulte, Shaked Flur, and Robert Norton-Wright. The sail instruction-set semantics specification language. 2017.

[37] David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of c. In *International Conference on Interactive Theorem Proving*, pages 99–115. Springer, 2012.

[38] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2:5, 2011.

[39] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems (TOCS)*, 34(1):1, 2016.

[40] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. Mips: A microprocessor architecture. *ACM SIGMICRO Newsletter*, 13(4):17–22, 1982.

[41] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[42] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap–the formally verified optimizing compiler compcert. In *SSS'17: Safety-critical Systems Symposium 2017*, pages 163–180. CreateSpace, 2017.

[43] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. Compcert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS2 2018-Embedded Real Time Software and Systems*, 2018.

[44] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[45] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

[46] 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 ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[47] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[48] L Lamport. The tla+ hyperbook.

[49] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[50] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009. ISSN 0168-7433. doi: 10.1007/s10817-009-9155-4. URL http://dx.doi.org/10.1007/s10817-009-9155-4.

[51] Xavier Leroy. The compcert c verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[52] 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.

[53] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.06: Documentation and user's manual*. PhD thesis, Inria, 2017.

[54] Developer Manual. Intel 64 and ia-32 architectures software developers manual, 2016.

[55] Ralph Melton, David L Dill, C Norris Ip, and Ulrich Stern. Murphi annotated reference manual. Technical report, Release 3.0. Technical report, Stanford University, Palo Alto, California, USA, 1996.

[56] Dale A Miller and Gopalan Nadathur. Higher-order logic programming. In *International Conference on Logic Programming*, pages 448–462. Springer, 1986.

[57] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. *SIGPLAN Not.*, 49(9):175–188, August 2014. ISSN 0362-1340. doi: 10.1145/2692915.2628143. URL http://doi.acm.org/10.1145/2692915.2628143.

[58] C Munoz. Batch proving and proof scripting in PVS. *NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, Report NIA Report*, (2007-03), 2007.

[59] Anthony Narkawicz, Cesar A Munoz, and Aaron M Dutle. The minerva software development process. 2017.

[60] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[61] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*,

volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. URL http://www.csl.sri.com/papers/cade92-pvs/.

[62] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag. URL http://www.csl.sri.com/papers/pvs-cav96/.

[63] Sam Owre. Random testing in pvs. 2006.

[64] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. N-version disassembly: differential testing of x86 disassemblers. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 265–274, 2010.

[65] Lawrence C Paulson et al. The isabelle reference manual. Technical report, University of Cambridge, Computer Laboratory, 1993.

[66] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 38–53, 2015.

[67] John Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[68] Herbert Schildt. *Java 2: the complete reference*. McGraw-Hill Professional, 2000.

[69] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[70] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482, 2013. doi: 10.1145/2491956.2462183. URL https://doi.org/10.1145/2491956.2462183.

[71] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[72] S Peter Song, Marvin Denman, and Joe Chang. The powerpc 604 risc microprocessor. *IEEE Micro*, (5):8–17, 1994.

[73] Richard Stallman et al. *Using GCC: the GNU compiler collection reference manual*. Gnu Press Boston, 2003.

[74] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 256–267, 2015.

[75] Éric Tanter and Nicolas Tabareau. Gradual certified programming in coq. In *ACM SIGPLAN Notices*, volume 51, pages 26–40. ACM, 2015.

[76] Simon Thompson. *Haskell: the craft of functional programming*, volume 2. Addison-Wesley, 2011.

[77] Ananthasayanam Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Amitava Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 430–444. IEEE, 2013.

[78] Muralidaran Vijayaraghavan, Adam Chlipala, Nirav Dave, et al. Modular deductive verification of multiprocessor hardware designs. In *Computer Aided Verification*, pages 109–127. Springer, 2015.

[79] Yusuke Wada and Shigeru Kusakabe. Performance evaluation of A testing framework using quickcheck and hadoop. *JIP*, 20(2):340–346, 2012. doi: 10.2197/ipsjjip.20.340. URL https://doi.org/10.2197/ipsjjip.20.340.

[80] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[81] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368, 2015.

[82] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 129–140, 2014.