On Reducing the Trusted Computing Base in Binary Verification

Xiaoxin An

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Thidapat Chantem
Dongyoon Lee
Haibo Zeng
Freek Verbeek

May 9, 2022
Blacksburg, Virginia

Keywords: Translation Validation, Disassembly Soundness, Binary Verification, Random Testing, Symbolic Execution, Bounded Model Checking
Copyright 2022, Xiaoxin An

On Reducing the Trusted Computing Base in Binary Verification

Xiaoxin An

(ABSTRACT)

The translation of binary code to higher-level models has wide applications, including decompilation, binary analysis, and binary rewriting. This calls for high reliability of the underlying trusted computing base (TCB) of the translation methodology. A key challenge is to reduce the TCB by validating its soundness. Both the definition of soundness and the validation method heavily depend on the context: what is in the TCB and how to prove it. This dissertation presents three research contributions. The first two contributions include reducing the TCB in binary verification, and the last contribution includes a binary verification process that leverages a reduced TCB.

The first contribution targets the validation of OCaml-to-PVS translation – commonly used to translate instruction-set-architecture (ISA) specifications to PVS – where the destination language is non-executable. We present a methodology called OPEV to validate the translation between OCaml and PVS, supporting non-executable semantics. The validation includes generating large-scale tests for OCaml implementations, generating test lemmas for PVS, and generating proofs that automatically discharge these lemmas. OPEV incorporates an intermediate type system that captures a large subset of OCaml types, employing a variety of rules to generate test cases for each type. To prove the PVS lemmas, we develop automatic proof strategies and discharge the test lemmas using PVS Proof-Lite, a powerful proof scripting utility of the PVS verification system. We demonstrate our approach in two case studies that include 259 functions selected from the Sail and Lem libraries. For each function, we generate thousands of test lemmas, all of which are automatically discharged.

The dissertation's second contribution targets the soundness validation of a disassembly process where the source language does not have well-defined semantics. Disassembly is a crucial step in binary security, reverse engineering, and binary verification. Various studies in these fields use disassembly tools and hypothesize that the reconstructed disassembly is correct. However, disassembly is an undecidable problem. State-of-the-art disassemblers suffer from issues ranging from incorrectly recovered instructions to incorrectly assessing which addresses belong to instructions and which to data. We present DSV, a systematic and automated approach to validate whether the output of a disassembler is sound with respect to the input binary. No source code, debugging information, or annotations are required. DSV defines soundness using a transition relation defined over concrete machine states: a binary is sound if, for all addresses in the binary that can be reached from the binary's entry point, the bytes of the (disassembled) instruction located at an address are

the same as the actual bytes read from the binary. Since computing this transition relation is undecidable, DSV uses over-approximation by preventing false positives (i.e., the existence of an incorrectly disassembled reachable instruction but deemed unreachable) and allowing, but minimizing, false negatives. We apply DSV to 102 binaries of GNU Coreutils with eight different state-of-the-art disassemblers from academia and industry. DSV is able to find soundness issues in the output of all disassemblers.

The dissertation's third contribution is WinCheck: a concolic model checker that detects memory-related properties of closed-source binaries. Bugs related to memory accesses are still a major issue for security vulnerabilities. Even a single buffer overflow or use-after-free in a large program may be the cause of a software crash, a data leak, or a hijacking of the control flow. Typical static formal verification tools aim to detect these issues at the source code level. WinCheck is a model-checker that is directly applicable to closed-source and stripped Windows executables. A key characteristic of WinCheck is that it performs its execution as symbolically as possible while leaving any information related to pointers concrete. This produces a model checker tailored to pointer-related properties, such as buffer overflows, use-after-free, null-pointer dereferences, and reading from uninitialized memory. The technique thus provides a novel trade-off between ease of use, accuracy, applicability, and scalability. We apply WinCheck to ten closed-source binaries available in a Windows 10 distribution, as well as the Windows version of the entire Coreutils library. We conclude that the approach taken is precise – provides only a few false negatives – but may not explore the entire state space due to unresolved indirect jumps.

This work is supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112090028 and contract N6600121C4028, and the US Office of Naval Research (ONR) under grants N00014-17-1-2297 and N00014-18-1-2665.

On Reducing the Trusted Computing Base in Binary Verification

Xiaoxin An

(GENERAL AUDIENCE ABSTRACT)

Binary verification is a process that verifies a class of properties, usually security-related properties, on binary files, and does not need access to source code. Since a binary file is composed of byte sequences and is not human-readable, in the binary verification process, a number of assumptions are usually made. The assumptions often involve the error-free nature of a set of subsystems used in the verification process and constitute the verification process's trusted computing base (or TCB). The reliability of the verification process therefore depends on how reliable the TCB is. The dissertation presents three research contributions in this regard. The first two contributions include reducing the TCB in binary verification, and the last contribution includes a binary verification process that leverages a reduced TCB.

The dissertation's first contribution presents a validation on OCaml-to-PVS translations – commonly used to translate a computer architecture's instruction specifications to PVS, a language that allows mathematical specifications. To build up a reliable semantical model of assembly instructions, which is assumed to be in the TCB, it is necessary to validate the translation.

The dissertation's second contribution validates the soundness of the disassembly process, which translates a binary file to corresponding assembly instructions. Since the disassembly process is generally assumed to be trustworthy in many binary verification works, the TCB of binary verification could be reduced by validating the soundness of the disassembly process.

With the reduced TCB, the dissertation introduces WinCheck, the dissertation's third and final contribution: a concolic model checker that validates pointer-related properties of closed-source Windows binaries. The pointer-related properties include absence of buffer overflow, absence of use-after-free, and absence of null-pointer dereference.

Acknowledgments

I would like to thank my advisor, Prof. Binoy Ravindran. Without his help and encouragement, I could not finish my Ph.D. program. I also would like to thank my co-advisor, Dr. Freek Verbeek, for all his efforts and guidance that are dedicated to my research. In addition, I would like to thank Dr. Giuliano Losa, Dr. Yakoub Nemouchi, and Dr. Amer Tahat. It is a great honor to have worked with them. Also, I would like to thank my committee members: Prof. Thidapat Chantem, Prof. Dongyoon Lee, and Prof. Haibo Zeng, for their constructive advice and feedback.

I would like to thank my friends and co-workers in the Systems Software Research Group: in alphabetical order, Joshua Bockenek, Anthony Carno, Ho-Ren Chuang, Jae-Won Jang, A K M Fazla Mehrab, Yihan Pang, Ian Roessle, Md Syadus Sefat, Cathlyn Stone, Mincheol Sung, and many others. Their selfless sharing of their life and opinions help me go through some tough times and face the setbacks in my research.

Finally, I would like to thank my parents and my brother. Their supports give me great courage to follow my dreams. Also, I would like to thank my husband, Qingyu Liu. The last two years have been especially tough due to COVID-19. Without Qingyu's encouragement, I do not know whether I could make it through. I would like to thank him for all his accompany, patience, and support.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

CFG control flow graph

DSV disassembly soundness validation

ISA instruction set architecture

LOC lines of code

OPEV OCaml-PVS equivalence validation

TCB trusted computing base

Chapter 1

Introduction

Many program verification and analysis tools that target binaries assume a trustworthy semantics of assembly instructions and a trustworthy disassembler. The tools build their analysis on top of these two assumptions, i.e., these are parts of the trusted computing base (TCB). To reduce the TCB and thereby increase the reliability of verification and analysis, this dissertation presents trustworthy semantics for assembly instructions and techniques to validate disassemblers, and shows that binary verification can be done on top of a validated disassembler.

1.1 Motivations

In recent years, a multitude of methods has been developed to verify the properties of binaries. These methods serve as important elements in many reverse engineering and related sub-disciplines such as decompilation [22], binary analysis [15, 31, 102], binary verification [100], and binary rewriting [11, 119]. For example, MAYHEM [17] exploited hybrid symbolic execution and index-based memory modeling to automatically detect bugs in executable files. SAGE [40] and S2E [20] are symbolic execution frameworks that are developed for binaries and can be used in different application contexts, including binary security verification.

1.1.1 The Trusted Computing Base of Binary Verification

In developing binary verification or analysis techniques, various assumptions have to be made, which form the TCB of the verification/analysis process. In general, the TCB must be reduced as much as possible. A large TCB means that there are many components that need to be trusted before the result of the binary verification technique can be trusted. Minimizing a TCB thus leads to more reliable verification. Considering the various assumptions made in the binary verification process, it is necessary to identify and validate some of the fundamental assumptions to reduce the TCB in binary verification.

Figure 1.1 provides an overview of the TCB when developing a binary verification technique. It shows four components that are generally trusted, i.e., assumed to be correct without further validation.

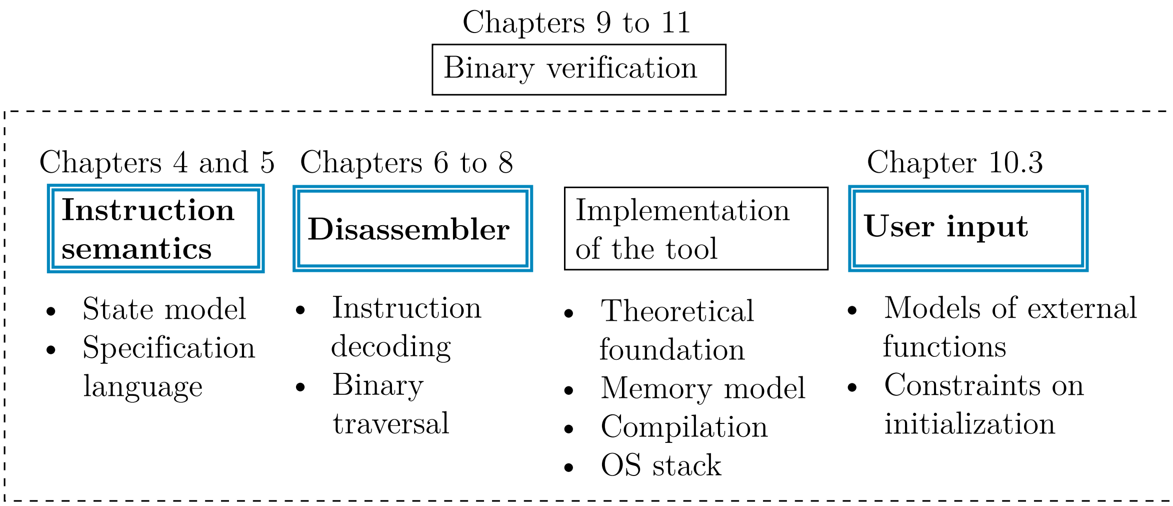

Figure 1.1: The composition of the TCB for binary verification.

Instruction Semantics. An assumption that is often implicitly part of the TCB in a binary verification effort is that a proper semantical model exists for all instructions in the binary. The semantics are dependent on a state model, i.e., which parts of the state are taken into account. Moreover, they are expressed in some specification language. Both the state model and the specification language effectively determine the preciseness of the semantics. Typically, the semantics specify the values of all registers and memory, but precise semantics may also provide information on flags, segment registers, deprecated floating-point operations, etc.

To build up a semantical model of assembly instructions in the binary verification tool, it may be necessary to translate instruction semantics to some specific language. This translation is often assumed to be reliable in the literature [48, 61].

The disassembler. A second assumption that is often implicit in a binary verification effort is the correctness of the disassembler. The input of a binary verification technique is a binary, written in machine code indicating the operations that the computer is performing during execution. The machine code is represented as byte sequences and has no well-formed formal semantics. Since the machine code is composed of byte sequences and is hard to understand, researchers have to translate the machine codes to assembly languages in a human-readable way and continue the analysis/verification of the assembly instructions. This translation, the *disassembly process*, is assumed to be trustworthy in many binary verification works [102, 113]. Typical parts of the disassembler that may introduce soundness issues are 1.) the instruction decoder (per individual instruction, translate a byte sequence into a humanly readable representation) and 2.) the way the binary is traversed and the way it is determined which addresses in the binary contain instructions.

Implementation of the tool. The tool itself is implemented in a programming language, and it is generally trusted that this implementation is correct. This part of the TCB encompasses the theoretical foundation on which the tool bases its verification; e.g., it can be based on model-checking [111] and state-space exploration [120], or on predicate transformations proven correct by Hoare logic [34]. The implementation also necessarily makes assumptions on the memory model: typically, separation must be assumed between different regions in memory. This component also encompasses the actual implementation, its compilation, and the OS stack on which the tool is run.

User input. Generally, any binary verification effort requires some form of user input. It may be the case that the tool must interactively be guided, but it may also be the case that the user is assumed to provide information on, e.g., semantical behavior of external functions or constraints on the initialization of the program under verification. Typically, the user input must be trusted, i.e., if the user does not provide proper information, then the verification effort becomes unreliable.

1.1.2 Motivation behind Minimizing the TCB

We thus argue the need for *building up trustworthy instruction semantics in a formal language*. We consider a specific translation from the specification language Lem [64] to the formal language of PVS [84]. This translation can be validated using various methods, such as testing. Testing is a widely used strategy to establish equivalence between two specifications. However, validating a translation by testing requires that both languages are executable. Some specifications can be either executable or non-executable, and the results of the non-executable specification cannot be directly computed. For example, in the Prototype Verification System (PVS) [79], PVSio [71], PVS's emulator utility, can only execute a subset of the functional specifications in PVS. This is a limitation of many theorem provers, not just PVS – their specification languages are designed to state and prove theorems but not execute. In fact, large subsets of many provers' powerful specifications are non-executable. This downside can be overcome by stating theorems on these specifications that capture the intended behaviors and proving them, mostly interactively – a highly labor-intensive effort. For example, verification of the CompCert compiler [55] using interactive theorem proving involved 100K lines of Coq proof [9] and six person-years of effort. The key challenge here is to find a methodology for translation validation of the Lem-to-PVS translation that provides more formal assurances than testing while being less human-labor intensive than manual verification through interactive theorem proving.

Moreover, we argue the need for *validation of the output of disassemblers*. Trustworthy disassembly is an essential component of the TCB in many reverse engineering and related sub-disciplines. A plethora of disassemblers exist [15, 85, 89, 102] that can recover assembly instructions from an executable binary. In many reverse engineering works, it is implicitly assumed that the disassemblers and the disassembly process are trustworthy. This premise holds true for both well-developed commercial and open-source disassemblers. For example, Ramblr [113] uses static binary rewriting to implement binary reassembling. The authors take angr [102] as the base platform to disassemble the binary and rebuild the program control flow graph (CFG), which implies that the correctness of Ramblr highly relies on the correctness of angr. As another example, Ghidra [89] is a state-of-the-art tool for decompilation. Ghidra's capabilities include control-flow reconstruction, type-inference, and pointer analysis, and all these functionalities are based on the assumption that disassembly is done correctly. Still, disassembly is not a solved problem: new techniques are developed based on machine learning [82], advanced heuristics, and inference methods, among others [85, 89, 102]. All existing disassemblers have soundness issues and may produce different results on the same binary. Instead of trusting a disassembler, we thus argue for validating its output.

An important goal of binary verification is to verify memory-related properties of closed-source binaries such as many Windows binaries. Examples of such properties include absence of buffer overflow, absence of null pointer dereference, and absence of use-after-free. These properties have a large impact on program security. For instance, in 2003, the SQL Slammer worm used a buffer overflow bug to infect a large number of machines running Microsoft SQL Server 2000, which caused a denial of service (DoS) on many hosts [103]. Null-pointer dereference took up 37.2% of all the memory problems in Mozilla and Apache [68]. As another example, from Microsoft Internet Explorer 6 to 11 [19], there exists a use-after-free vulnerability named CVE-2014-1776, which can be used to cause a DoS attack.

We thus argue that there is a need for *a binary verification tool for Windows executables, where no ground truth is available*. Since no source code, debugging information, or any other form of ground truth is available, we argue that such a verification tool must be built on top of a reduced TCB. For example, the tool must use validated disassembled instructions.

1.2 Challenges

To validate the translation from a reliable instruction semantics written in one language to an instruction semantical model represented in another language, a fundamental problem is how to define the *soundness* of the translation between a language without well-defined semantics (e.g., x86 machine code) and a language with well-defined semantics (e.g., x86 assembly language). Since the source and destination languages do not have formally-specified behaviors, developing a conformance relationship between programs written in the two languages is challenging. In some situations, a translation between two languages is sound if the corresponding two programs always produce the same output for the same input. In other words, they have I/O equivalence [51]. However, for some languages that support non-executable semantics, I/O equivalence cannot be established, and the soundness definition of the translation is still a challenge.

The choice of the methodology used to validate the translation is another challenge. Refinement proof [44] is a rigorous method for translation validation. However, it requires a formal model of the source and target languages. Often, it is challenging to build such formal models, as in many cases, the semantics of the two languages cannot be mapped to each other one-to-one. Moreover, if one language does not have well-formed semantics, extracting a formal model for it is another challenge. In such situations, testing [28], especially random testing [35], is an effective method and can detect inconsistencies. However, as previously discussed, testing is not applicable in situations when the destination language is not executable, as is the case with PVSio [71], PVS's emulator utility.

The validation of the disassembly process, which is another step of binary verification, is also challenging. Disassembly is by its very nature inherently an *untrustworthy* process. It is an undecidable problem [87, 115] due to multiple reasons, such as variable-length instruction encoding and mixed instruction and data. In a context where only the binary is available (e.g., software with proprietary code), there is *no ground truth* as to what the "correct" assembly instructions are. State-of-the-art and mainstream disassemblers such as objdump [38], Hopper [50], and IDA Pro [52] suffer from issues when, e.g., instructions are overlapping, data and instructions are mixed, indirect jump/call targets are unresolved, or a security vulnerability leads to unexpected control flow. Many errors have been reported for these disassemblers, such as incorrectly recovered instructions and incorrectly assessing which addresses belong to instructions and which to data [7, 70].

Even though the foregoing challenges must be solved, in implementing a binary verification tool to verify closed-source binaries, three fundamental challenges have to be overcome: i) *unbounded loops*, ii) *pointer-aliasing*, and iii) *external functions*. The handling of unbounded loops is an essential problem for any verification effort based on symbolic execution [56, 94], as it may cause state-space explosion or non-termination. The problem with pointer-aliasing is that during symbolic execution, it may not be known that two given pointers refer to overlapping or separate regions in memory. As a result, it is difficult to provide an accurate step-function that describes the state change induced by a memory write operation. Computing pointer aliasing relationships becomes more difficult due to the lack of typing information in a binary since variables are simply regions in memory (in a binary). Finally, a closed-source binary typically calls various external functions whose machine code is not accessible until linked at run-time. Thus in the static analysis process, the functionality and calling convention of these external functions are unknown. Note that these problems also manifest in verifying open-source binaries, but they are particular challenges for closed-source binaries – the lack of sources prevents inferring useful information from the source code, such as loop bounds, variable types, and identification of external functions.

The aforementioned challenges in verifying closed-source binaries exacerbate for Windows binaries. Since many Windows binaries are closed-source, the verification process cannot use the source code as the ground truth. Whether a detected memory-related negative is a true error is undecidable. Windows binaries are often compiled with aggressive levels of optimization, which makes both loop- and pointer-analysis difficult. Moreover, in Windows binaries, external functions may have mixed calling conventions even within the same binary, which is allowed in the Windows ABI [1]. Such mixed calling conventions aggravate the verification process.

Motivated by these challenges, we aim to reduce the TCB of a binary verification process (Figure 1.1) via different methods. In Chapters 4 to 5, we target the translation from Lem to PVS. Chapters 6 to 8 target disassembly validation. Chapters 9 to 11 provide a method for Windows binary verification with minimal TCB. Specifically, Section 10.3 discusses how the amount of information requested from the user is minimized.

1.3 Dissertation Contributions

This dissertation presents three research contributions. The first contribution is a technique for validating the equivalence relationship within an OCaml-to-PVS translation. The motivation for validating this translation lies in the requirement for translation from the Sail language to PVS. The Sail language [43], which is a first-order imperative language, has been used to describe the semantics of ISAs such as x86, ARM, RISC-V, and PowerPC [43]. 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 [43]. While translators from Sail to theorem provers such as Isabelle/HOL, HOL4, and Coq exist [43], one to PVS [79] does not. We develop a Sail-to-PVS translator to translate the semantics of ISAs specified 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; we then employ the executable feature of OCaml to validate the OCaml-to-PVS translation. If the equivalence between OCaml and PVS is validated, the Sail to PVS translation is validated. We, therefore, develop 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 [90], proof checking [77], and batch proving [71] to solve the validation problem.

The dissertation's second contribution is a technique for validating the soundness of the disassembly process. Disassembly is a translation from machine code in a binary to assembly code. Machine code has no well-defined semantics, whereas assembly code has. We set up a limitation for this problem in that we assume that we do not have access to the original assembly code, and thus we do not have the assembly code as the ground truth 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 backwards compatible. This limitation raises the problem of how to determine the ground truth for soundness validation while requiring that the correctness of the disassembly must be validated without assembly code. We focus on the widely used Intel x86

ISA [53], which is another challenge since the x86 ISA has a large number of instructions with variable lengths, and the documentation does not provide clear specifications for some instructions. Our validation technique uses a soundness notion that is defined using a transition relation defined over concrete machine states and the reachability of addresses in the binary from the binary's entry point. Since computing this relation is undecidable, our technique uses over-approximation by preventing false positives and allowing, but minimizing, false negatives.

The dissertation's third and final contribution is a concolic model checker that detects memory-related properties, including absence of buffer overflow, absence of use-after-free, and absence of null-pointer dereference in closed-source Windows binaries. Violation of these memory properties is pervasive and often the source of many security exploits. Verifying them for closed-source Windows executables is particularly challenging as such executables often have different sources and use mixed calling conventions. We develop a formal definition of these properties and develop a concolic model checker that uses those definitions to detect the violation of these properties.

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 methodology combines testing and proving to validate properties, which requires short development cycles and supports validation using a formal verification language (i.e., PVS). Our methodology, 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 rules we have developed for an intermediate type system that captures the commonality of OCaml and PVS types. The test cases are directly executed on the OCaml program and are also used for constructing a large number of test lemmas on the PVS specification. We represent the test lemmas as equations with test cases on the left-hand side and the execution results on 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 PVS feature called proof strategies [71]. The results of the proofs are then employed to establish equivalence. Figure 1.2 illustrates OPEV.

We demonstrate OPEV by using it to validate a manually implemented OCaml-to-PVS translation and a manually implemented Sail-to-PVS parser. The Sail-to-PVS parser includes 2,763 LOC and is used to translate 7,542 LOC of Lem code to 10,990 LOC of PVS implementation. OPEV generates and proves 458,247 test lemmas for these two case studies and detects 11 errors. The development of OPEV takes three person-months, and the effort to develop and validate the translator takes five person-months.

OPEV's central contribution is the semi-automatic test-and-proof methodology for validating

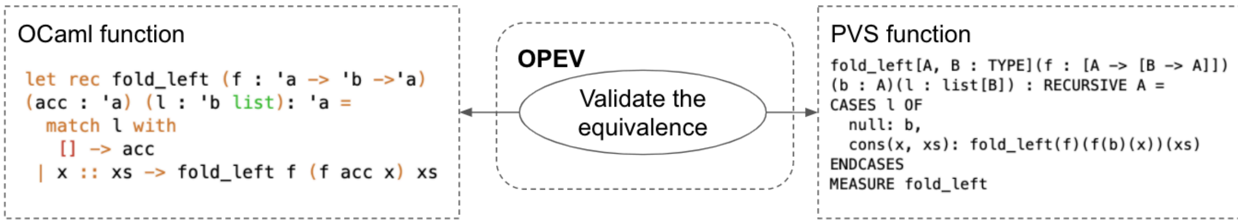

Figure 1.2: Equivalence validation for OCaml and PVS.

translators supporting non-executable specifications. In principle, the OPEV methodology can be applied to any pair of languages where one has non-executable semantics.

1.3.2 Disassembly Soundness Validation

We develop a formal definition for the soundness of disassembly. Our soundness concept uses a transition relation defined over concrete machine states: a binary is sound if, for all addresses in the binary that can be reached from the binary's entry point, the bytes of the (disassembled) instruction located at an address are the same as the actual bytes read from the binary. Thus, a disassembler is unsound if there is a reachable (disassembled) instruction whose bytes are not the same as those in the binary. Since computing this transition relation is undecidable, we use over-approximation by preventing false positives, i.e., the existence of an incorrectly disassembled reachable instruction but deemed unreachable, and allowing, but minimizing, false negatives, i.e., the existence of an incorrectly disassembled unreachable instruction but deemed reachable.

Based on this definition, we implement a tool called DSV, for "Disassembly Soundness Validation," to validate whether a binary has been soundly disassembled or not. As illustrated in Figure 1.3, DSV takes a binary file and the assembly file disassembled from the binary file as inputs, generates "sound" or "unsound" as output, and reports all the "unsound" disassembled instructions. Essentially, DSV performs a recursive traversal starting at the binary's entry point while validating all reached instructions. DSV's key characteristic is that it does *not* assume a ground truth; in other words, DSV does not presume the availability of source code or debug information.

DSV over-approximates the semantics of the binary under investigation in two ways. First, the semantics of various instructions are over-approximated by treating their effects on certain state parts as unknown. Second, the jumps and paths that can be traversed at run-time are statically over-approximated. DSV needs to deal with three key problems: i) unbounded loops, ii) pointer aliasing, and iii) indirect-branch instructions. In order to deal with loops, we employ *bounded model checking* (BMC) [13]. To handle the pointer aliasing problem and indirect branches, we use *concolic execution* [98], which is a combination of concrete and symbolic execution.

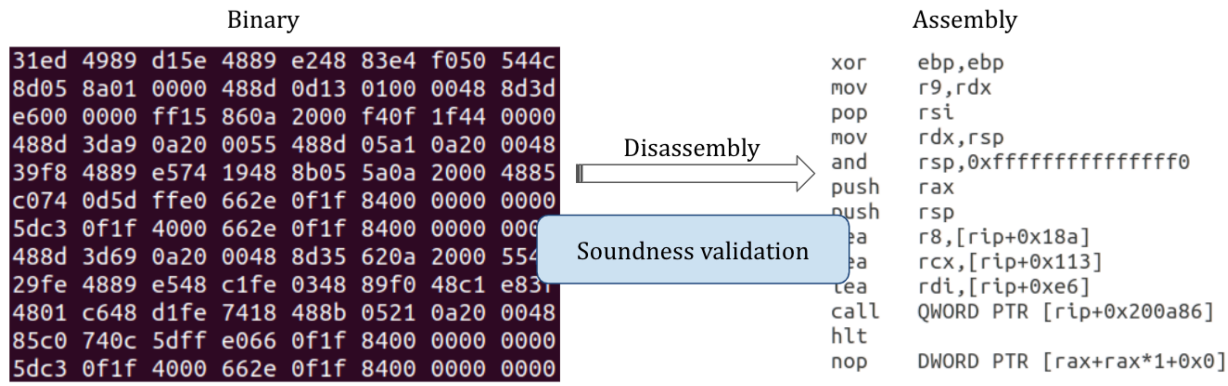

Figure 1.3: Soundness validation for disassembly.

We apply DSV to all the binaries of the GNU Coreutils library for eight different disassemblers. Soundness issues are found in each of them. Some examples include:

1. Incorrectly recovering instructions, e.g., Ghidra [89] disassembles `49 0f a3 c8` to `bt rax,rcx`, while the correct result is `bt r8,rcx`;

2. Incorrectly recovering immediate values in operands, e.g., Dyninst [11] translates `48 b8 ff ff ff ff ff` to `mov rax, 0x4611686018427387903`, however, the valid instruction is `movabs rax,0x3fffffffffffff`;

3. Missing instructions due to under-approximating indirect control flow transfers.

DSV's contribution consists of:

1. A formal definition for the soundness of disassembly.

2. An automated methodology for validating whether the output of a black-box disassembler is sound with respect to a binary.

3. The application of this methodology to 102 binaries of Coreutils, each for eight different disassemblers: angr 8.19.7.25 [102], BAP 1.6.0 [15], Ghidra 9.0.4 [89], objdump 2.30, radare2 3.7.1 [85], Dyninst 10.2.1 [11], IDA Pro 7.6, and Hopper 4.7.3.

1.3.3 Verifying Pointer-Related Properties of Closed-source Binaries

We present WinCheck, a model checker for closed-source binaries, such as Windows executables and libraries. The model checker is tailored for properties pertaining to pointers. The set of properties includes security-related properties such as use-after-free, buffer overflow, and null-pointer dereference.

Similar to DSV, WinCheck also needs to tackle the challenges of unbounded loops and pointer aliasing, and additionally, external functions. WinCheck approaches these three challenges as follows. Loops are dealt with using *bounded* model checking. Bounded model checking is applicable even in cases where no loop-invariants can be established. The cost is that it may lead to false positives in case the bound is actually hit. Pointer-aliasing is dealt with using *concolic* model checking. State parts are kept as symbolic as possible, but pointers are always concrete. For example, the stack pointer and the return values of `malloc` are kept concrete. This solves the pointer-aliasing problem since that problem is decidable if all pointers are immediate values. External functions are dealt with by interactively requesting the user for necessary information.

The model checker is based on a *traceback* system: as soon as a pointer-relation becomes symbolic, the model checker traces back to the root of the issue. If this is due to an external function, the user is asked for information about that function. Effectively, the user is interactively asked for a very limited amount of information regarding the effects of external functions on pointers in a given state (for example, function `malloc` returns a fresh pointer).

We applied WinCheck to several closed-source Windows 10 binaries in PE format, as available in a standard Windows distribution, as well as the Windows version of the entire Coreutils library. WinCheck supports 64-, 32-, and 16-bit ISAs. We found that WinCheck's concolic nature produces only a small amount of false negatives. An example of a false negative is a read from uninitialized memory, which is not a true negative if the read data is never used (this occurs, for example, during stack probing). However, due to WinCheck's boundedness as well as unresolved indirect jumps (control flow transfers computed at run-time), only a part of the binary is explored.

To verify the absence of memory-related issues on binaries, different methods have been under investigation for decades. These methods vary from random testing (often under-approximative, i.e., subject to false positives) to formal verification (often over-approximative, i.e., subject to false negatives). We argue that WinCheck is the first model checker that satisfies all of these characteristics:

1. applicable to Windows executables without needing to model the system under verification;
2. no need for specifications, definitions of properties, or source-code level assertions;
3. the required interaction is limited to asking the user for specific necessary information.

Note that the *implementation* of WinCheck is part of the TCB (see Figure 1.1), and we do not target removing the *implementation* from the TCB. Specifically, our method (and also tool) for Windows binary verification called WinCheck itself has not been verified.

WinCheck assumes that stack, heap, and global data are separate memory spaces and that malloc returns a fresh region separate from existing regions. Concurrency is scoped out, i.e., only single-threaded binaries are supported.

In this dissertation, we use the terminologies of *validation* and *verification* in a different way than how it is often used [108]. To make a distinction, the terminology *validation* indicates the act that checks whether an input/output pair is correct. Meanwhile, *verification* illustrates the act that checks whether the verifying process works correctly for all inputs. Although WinCheck verifies the pointer-related properties in an under-approximative way, which means WinCheck is subject to false positives, WinCheck endeavors to detect all the possible pointer-related errors for all the inputs. Thus, we use the terminology *binary verification* to describe WinCheck.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents the background of symbolic execution, formal verification, and model checking.

Chapter 3 presents past and related work in each of the dissertation's problem spaces and compares and contrasts them with the dissertation's contributions.

Chapter 4 presents the OCaml-to-PVS equivalence validation work. We introduce the proof automation of PVS and the corresponding case studies of OPEV in Chapter 5.

Chapter 6 presents the soundness definition of disassembly and Chapter 7 presents the implementation of the DSV disassembly soundness validation tool. DSV's case studies are presented in Chapter 8.

In Chapter 9, we introduce the pointer-related properties that can be detected by WinCheck, in a formal way. Chapter 10 presents WinCheck's implementation details. WinCheck's experimental results and analysis are presented in Chapter 11. Chapter 12 concludes the dissertation and identifies future work.

The chapters presenting this contributions are based on OPEV [4], DSV [5], and WinCheck [6].

Source Code Availability

Artifacts of the OPEV methodology are open-source and publicly available at: https://ssrg-vt.github.io/Renee/.

The complete source code, benchmarks, and experimental results for DSV are open-sourced and available at the project website: https://ssrg-vt.github.io/DSV. The source code artifact is archived with a DOI link at: https://doi.org/10.5281/zenodo.6380975.

Chapter 2

Background

In our work, we refer to different kinds of formal methods and apply testing, semi-automatic proofs, symbolic execution, and bounded model checking 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 [60]. 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. More specifically, each value that cannot be resolved by static analysis of the code is denoted by the symbolic value. By evaluating all the generated execution paths, 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 the concrete input value. However, we cannot declare that this assertion is *true* in any case. If we apply symbolic execution to the code, the symbolic execution generates an unlimited number of execution paths since the code contains loops, and the termination condition is a symbol.

Moreover, if the symbolic path constraint is not solvable or cannot be solved efficiently, inputs 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 [98], are developed to solve the problem. Moreover, to reduce the number of generated paths, algorithms that eliminate infeasible paths during the symbolic execution

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);
}
```

are developed [2].

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 [61], 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 generators, such as Isabelle/HOL theorem prover [74], 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 [9], HOL4 [42], Isabelle/HOL [83], and PVS [79], 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 [61], an OS microkernel. Coq has been applied to analyze Verdi [116], a distributed system, and Compcert [55], 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 [110], if the number of processors in a multiprocessor system is parameterized, which means any number of processors is acceptable in 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.

In recent years, formal verification using interactive theorem provers has been extensively applied to software and hardware systems, with the increasing requirement of system correctness and soundness. For example, in the software field, Compcert [65], a C compiler, has been verified using the Coq theorem prover; and Ridge et al. [88] presented a model of the behaviors that are permitted by the SibylFS file system. In the hardware field, Vijayaraghavan et al. [110] modeled, refined, and proved a multiprocessor hardware system, which consisted of a parameterized number of processors and parameterized level of cache hierarchies, using the Coq theorem prover.

Even further, Klein et al. [61] have employed Isabelle/HOL to formally verify the seL4 microkernel from the specification of the model to the low-level C implementation, which demonstrates 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 [61] provides high-level assurance of the functional correctness of an OS kernel in the 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 indicates that the formally verified kernel could also achieve high performance. The complete functional correctness of seL4 was verified. Besides, the researchers assumed the correctness of the C compiler and the real hardware, which are the TCB in the verification. There were some limitations during the verification of seL4. For instance, seL4 only allowed 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 a 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 employs a particular language to construct the model of the system, describes the specifications of the requirements in the form of formulas, and analyzes 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 that can be verified using temporal logic. Besides, different model checkers analyze distinct properties. For example, CBMC [62] could verify array bounds, pointer safety, exceptions, and user-specified assertions in C code; and TLC [63] checks the specification of some simple system constructed using TLA+ or PlusCal language.

Explicit-state model checkers, such as Murphi [69], TLC [63], and SPIN [49], 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 [21] and NuSMV [24], have better performance in analyzing 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 have to be translated into transitions between sets.

As the technique develops, automation becomes a key requirement in model checking. For example, CBMC [62] is applied to test C programs and verify predefined properties automatically. However, in some cases, because of the infinite state space and undecidability of the problem, such as whether a C program would terminate or not, it is impossible to apply automatic proving. Handwork is still necessary in such cases. Besides, even though model checkers can be employed to prove some rather complex systems, 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 the model-checking technique. Model-checking technique is widely applied to verify many software and hardware systems.

Linux Virtual File System [36]

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 the C source code of the implementation of VFS together with some manually inserted code. Then, SPIN [49] and SMART [21], 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. The modeling process cannot be executed completely automatically because of some specific elements, such as dynamic memory allocation, macros, and inlined assembly. Thus, the Kernel Function Trace tool is selected to get traces from the executions of a Linux kernel, and manual examination is also adopted to assemble an abstract VFS model in C language.

The simulation of the model is implemented using the SPIN model-checking technique [49] for the following reasons. First, the 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 competencies and accepts assertions inserted while running. Thus, a 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 of verifying the model.

Therefore, SMART [21], a symbolic model checker, is introduced to verify the model. The verification using SMART is based on Petri nets, and the VFS model is translated into a Petri net and then is analyzed using SMART. In the new model, various VFS variables are parameterized and symbolized as Petri net nodes, and calls are depicted as transitions. The main properties of the VFS model, which are verified using the SMART checker, are deadlock-freedom and data integrity which includes three elements: allocation, reference, and structural properties.

Hypervisor Framework [109]

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 properties. 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 [62]. This framework is evaluated by being compared with other general hypervisors, and the evaluation results show that the performance of XHMF is as outstanding as other popular hypervisors.

The verification of the structure is implemented by CBMC, 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 reduces the handwork.

2.3.2 Concolic Model Checking

Concolic model checking is a technique that combines concolic execution and model checking together to check certain system's properties. It holds the characteristic of model checking, such as exploring the state space of a model to check specific properties of a system. Meanwhile, the concolic execution process enables the concolic model checker to verify certain properties in a decidable way. For example, if the memory address in a system model is enforced to be concrete while the other parts are symbolic, then certain challenges, such as pointer-aliasing problems, can be solved, which facilitates the validation of memory-related properties.

The combination of different software verification techniques has been extensively used in recent years. For instance, DART [39] and CUTE [98] used concolic testing to generate test inputs for C programs, while jCUTE [97] applied concolic testing to test concurrent Java applications. Meanwhile, JPF [111] is an explicit-state model checker that combines model checking and testing together to analyze Java programs. ExpliSat [8] also employed the concolic model-checking technique to validate specifically designed properties for C programs. The hybrid methodology, such as concolic testing and concolic model checking, gets the advantages of different techniques and enables the tools built upon the hybrid method to carry out the software verification process in a flexible way.

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 a disassembly process. Then some techniques regarding verification of pointer-related properties are illustrated in Section 3.3.

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 [55, 66] 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 [54]. However, the formal proofs of CompCert did not cover the correctness of the formal specifications of C and assembly [66]. In addition, it took six person-years of effort and involved 100,000 lines of Coq code [55].

In [100], the authors show that the seL4 source code [96] 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 the formal semantics that is 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 [61] took 8 person-years; the seL4 total verification effort [61] is more significant and took $\sim$20 person-years.

In contrast with *compiler verification* and *refinement proofs*, OPEV is a lightweight 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 is illustrated in Table 3.1.

OPEV also differs from some other testing-based lightweight verification techniques. For

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

Feature	sel4 [61]	CompCert [66]	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

Table 3.2: Comparison between OPEV and lightweight formal verification approaches.

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

instance, Haskell's QuickCheck mechanism [27] 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 [26] checks program annotations based on randomly generated test suites. Similar methods exist for theorem provers. For example, QuickCheck [107] and Nitpick [14] for Coq and Isabelle/HOL use random testing [112] 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 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 [30] to assist in proving these properties or reaching a counterexample [78]. 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.

The closest work to OPEV is MINERVA [73], which provides a practical approach to produce high assurance software systems using model animation on mirrored implementations for verified algorithms [73]. 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

We first discuss the main approaches to disassembly. Then, the approaches for the validation of disassembly are discussed.

3.2.1 Disassembly Techniques

Linear sweep and recursive traversal are the major techniques behind the binary disassembly process. PSI [119] and objdump [38] are typical linear-sweep disassemblers. These disassemblers handle the byte sequences in the binaries sequentially. Linear-sweep disassemblers have superior performance under certain circumstances. For example, some linear sweep disassemblers fulfill a 100% correctness on SPEC CPU 2006 benchmarks generated by gcc [37] and clang [7]. However, linear sweep disassemblers have poor performance in handling special situations such as overlapping instructions, inline data, and jump tables.

On the other hand, disassemblers such as IDA pro [52], Dyninst [11], Ghidra [89], and Hopper [50] are implemented using recursive traversal. These disassemblers decode the instructions following the execution path of the sequential and branching instructions and try to resolve the indirect jump addresses. Essentially, they reconstruct the *control flow* on-the-fly in order to perform disassembly. Recursive traversal handles overlapping instructions and inline data in a more reliable way than linear sweep disassemblers.

However, recursive traversal presents a crucial challenge, which is how to resolve indirect-branch addresses. The implementation of jump address resolving algorithms in various disassemblers leads to different performances. Researchers apply program slice [23] to recover jump tables from binary files. Meanwhile, constant folding and propagation are employed to resolve indirect branches in many applications [33, 58]. With this method, constants are propagated in a block, and comments with concrete targets are added to specific calls that call these constants. Furthermore, Kinder et al. [57] combined over-approximation and under-approximation together to construct indirect branches. Schwarz et al. [95] proposed a technique based on relocation information to collect possible indirect jump targets, and all these possible addresses were visited to advance the disassembly process.

Other than the traditional disassembly techniques, disassembly based on machine learning is gaining traction. Wartell et al. [114] implemented a machine learning-based approach to discriminate code from interleaving data, which is a crucial challenge in disassembly. Besides, though there are no full-fledged machine-learning-based disassemblers, a project named MLDisasm [105] was in development. MLDisasm was designed to serve as a disassembler and was developed based on LSTM neural network. A major challenge for MLDisasm was how to determine the instruction boundaries. Though MLDisasm is not a complete off-the-shelf disassembler, it provides another direction for future disassembly techniques.

3.2.2 Soundness Validation

Andriesse et al. [7] checked the false positive and false negative rates for nine mainstream disassemblers using SPEC CPU2006 and Glibc-2.22 as the benchmarks. The researchers gave a comprehensive comparison between different disassemblers on five critical criteria, including instruction recovery, function starting address relocation, function signature restoration, control flow graph (CFG), and callgraph reconstruction. They used the ground truth information derived from LLVM analysis, DWARF debugging information, and some manual ancillary work. These ground truths provided critical information for the five criteria.

Paleari et al. [80] developed a methodology called n-version disassembly to apply differential analysis to validate the correctness of different x86 disassemblers. The writers employed various disassemblers to recover the instruction from the same string of bytes and compared the results to find out the divergences. This paper validates the correctness of single-instruction disassembly, whereas our paper focuses on a complete disassembly process.

Pang et al. [81] manually evaluated the code base of various disassemblers and discussed the algorithm and heuristics used by these disassemblers. They also studied 3,788 binaries from different sources on nine mainstream disassemblers to evaluate the instruction recovery, cross-reference accuracy, function starting point, and CFG construction. They reported incorrectly disassembled cases existing in these disassemblers. The ground truths were automatically collected in the compiling and linking procedures when generating binaries with a method similar to the technique used by Andriesse et al. [7].

In DSV, a major concept is the reachability of instructions, which is implemented using an over-approximative abstract transition relation. Similarly, Kinder et al. [59], in 2009, proposed an abstract transition relation for over-approximative control flow construction. However, this work did not handle indirect jumps. A subsequent work, presented by Kinder et al. in 2012 [57], introduced a solution to handle indirect jumps. This solution, however, alternated between over- and under- approximation to construct control flow.

3.3 Property Verification Techniques

To validate the memory-related security properties, different methods have been under investigation for decades. We here distinguish static verification from runtime monitoring.

3.3.1 Static Property Verification Tools

Table 3.3 provides an overview. We divide the property verification algorithms into four major categories: model checking, testing, symbolic execution, and interactive theorem proving.

Table 3.3: Comparison between WinCheck and other model checking, symbolic execution, and testing tools.

Tool	**Type**	**Input language**	**Properties**
WinCheck	CMC	Windows executables	Pointer-related
TLA+	MC	TLA+ specification	LTL
SPIN	MC	Promela specification	CTL*
CPAChecker	MC	C programs	C assertions
NuSMV	SMC	SMV model	CTL & LTL
SAGE	SE	Windows applications	Abnormal termination
KLEE	SE	LLVM	N/A
MAYHEM	SE	Binary	N/A
BinSec	SE	Binary	N/A

(S/C)MC = (Symbolic/Concolic) Model checking
SE = Symbolic execution

Model Checking

Model checking is a technique that verifies a given property of a program against a finite-state model of the program. The model and the property of the program are typically formulated using specific model-checking tools, such as TLA+ [118], SPIN [49], UPPAAL [10], NuSMV [25], or CPAchecker [12]. Model-checking techniques are able to verify whether a program satisfies certain liveness or safety properties once the model has been formulated.

A key challenge in model checking is to derive a proper model. The input for a model checker typically is on a high level of abstraction (e.g., PROMELA or timed automata). In the case of a closed-source binary, there exists no reliable method to derive such a high-level model from the low-level machine code. WinCheck solves this issue by being directly applicable to the machine code of the executable. The trade-off is that WinCheck is tailored to specific properties and does not generalize to a complete logic of temporal properties.

Testing

Testing techniques are grandly employed in software development to validate certain properties. They provide concrete values as inputs to binaries to expose various problems. Different testing methods, such as random testing [18, 35, 45, 46], search-based testing [47], or fuzz testing [40], develop various strategies to increase the testing coverage. Unlike model-checking techniques, the testing inputs are concrete and finite, which means the testing inputs may not cover all the paths of the testing. How to enlarge testing coverage is a fundamental challenge for any testing technique.

Symbolic Execution

Symbolic execution is another extensively used technique to check properties in programs [60]. Symbolic execution uses symbolic, rather than concrete, values as inputs to execute the program. Symbolic execution generally aims at providing over-approximative results since the inputs are symbolic and all the paths are covered. However, a complete symbolic execution procedure on a large program is challenging because of the state/path explosion problem. Some whitebox fuzz testing techniques, e.g., SAGE [40], are developed based on advanced symbolic execution tools. SAGE is employed in detecting security-related errors in large and complex binary applications [41].

Interactive Theorem Proving

In 2000, Xu et al. [117] developed a system to analyze certain memory security properties in stripped SPARC executables. This technique was built upon theorem-proving. They used an induction-iteration method [104] to generate loop invariants. Myreen et al. developed decompilation-into-logic [72], which is a technique for lifting machine code into a representation in Higher-Order-Logic so that it can be interactively reasoned over in a theorem prover. Klein et al. used a refinement-based approach to verify the binary of the seL4 microkernel [61, 100].

Summary

WinCheck is directly applied on Windows executables and does not require a further step to lift the binary code to models written in a certain language. In contrast to CPAchecker [12] and KLEE [16], WinCheck does not require source code. Moreover, WinCheck does not require the formulation of properties, be it in the form of specifications written in CTL*, source-level assertions, or manual guiding of a symbolic execution engine.

WinCheck ensures that – in its concolic execution – memory addresses are always concrete. This solves the pointer aliasing problem that symbolic techniques inherently suffer from. For example, when KLEE is given a program with two pointers as parameters, it will simply assume that these pointers do not alias. This assumption may produce false positives, as paths may be missed. In contrast, WinCheck carries out concolic execution from the entry point of the executable, ensuring a concrete memory model while leaving the remainder of the state as symbolic as possible. It thereby provides a novel trade-off between 1.) automation and ease-of-use, 2.) scalability, and 3.) applicability.

3.3.2 Runtime Property Verification Tools

Runtime monitoring of software without source code is an open challenge. Multiple techniques have been proposed to detect memory security errors *with source code or debugging information available* at runtime. Intel MPX [75] is a hardware-based solution that serves as an Intel ISA extension to refrain from some memory security violations. Intel MPX is promising in that it supports a guarantee against errors such as buffer overflows. AddressSanitizer [99] implemented the trip-wire approach in a compiler, which could be used to detect buffer overflow and use-after-free bugs. CRED [91] and SAFECode [29] employed object-based methods to guarantee that the operations on pointers do not modify the objects to which the pointers refer. Both CRED and SAFECode can be applied to detect buffer overflow. While CRED maintained a high-performance overhead, SAFECode skipped certain buffer overflows in the same pool, which is partitioned by SAFECode using pointer analysis. For all of these techniques, source code is necessary for the program detected, which is different from WinCheck. In contrast, MAI [67] supports runtime detection of fine-grained memory-related errors on binaries. Runtime strategies inherently deal with the question of what to do when a violation is encountered. In contrast, a static approach such as WinCheck aims at showing a violation cannot occur at the very beginning.

Chapter 4

OPEV: OCaml-to-PVS Equivalence Validation

In this chapter, we introduce the OCaml-to-PVS equivalence validation (OPEV) methodology that builds up trust between the translated OCaml code into PVS. The translation is carried out automatically (for a subset of OCaml) or manually. Moreover, the translation is error-prone since these two languages are of different natures, where OCaml is a functional programming language and PVS is a language for formal verification.

The overall workflow of the OPEV methodology is presented in Section 4.1. In Section 4.2, we introduce 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; and the proofs of the generated test lemmas are shown in Section 4.4.

4.1 OPEV Workflow

We use Figure 4.1 to demonstrate the overall workflow for OPEV. As can be seen, we use an intermediate type system, which is introduced in Section 4.2, to cover the commonality of the type system between OCaml and PVS languages. The intermediate type system is confined to a subset of the entire OCaml and PVS type system. For each of the functions written in PVS or OCaml languages, OPEV parses the sources to formulate an intermediate type annotation. Then OPEV generates test cases, based on the generating rules introduced in Section 4.3, for each of the functions. OPEV executes the OCaml test cases to get the results, translates the OCaml outputs into PVS representation, and employs the test cases and the parsed outputs to construct test lemmas in PVS. The test lemmas serve as `test oracles`, which are automatically proved using generic PVS proof strategies that are manually implemented. If the test lemmas are validated to be false, we realize that there exist mismatches in the OCaml-to-PVS translation. Then we inspect the test cases and detect the reasons.

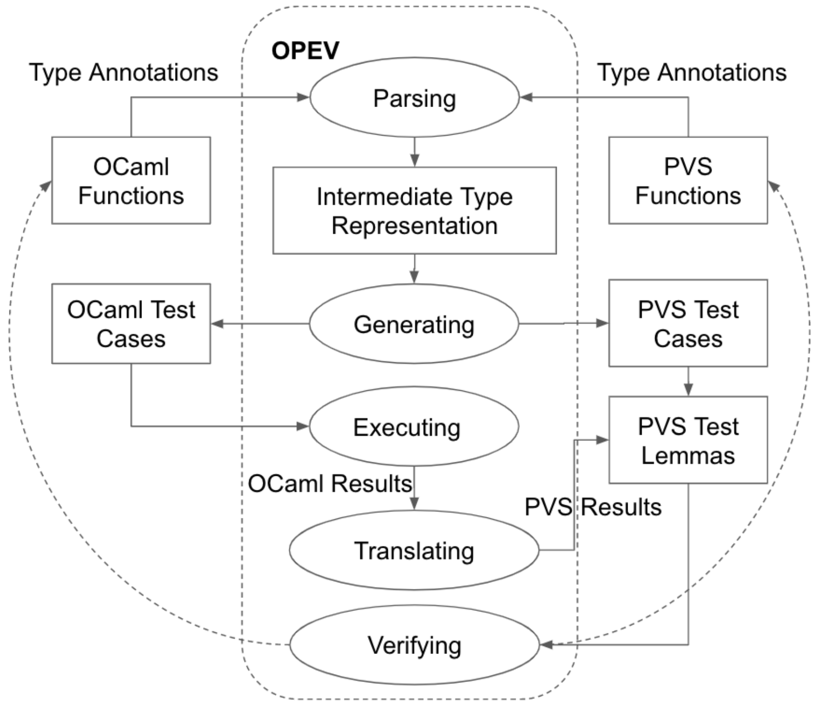

Figure 4.1: The OPEV workflow.

4.1.1 Extensibility

OPEV has already covered 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 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 extracted from the *res* variable, which has the value [8; 6; 1]. Then OPEV parses the result according to the return type of function rev and composes a PVS test lemma, such as th_rev as shown in Listing 4.2.

Listing 4.1: The definition of a PVS rev 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 PVS test lemma for rev function.

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

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

4.1.2 Non-Executable Semantics

We construct PVS test lemmas rather than directly executing the test cases in PVS because the semantics of some PVS functions are non-executable. Most of the functions with set-theoretic semantics in PVS are non-executable, including relational specifications, which are represented as predicates on sets in PVS. For example, as shown in Listing 4.3, the semantics of the function filter is non-executable. This is because the filter function introduces what kind of elements should be included in the result set after the execution of the function but does not indicate the steps of how to execute the function in PVS executable syntax.

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)}
```

Meanwhile, in PVS, functions with non-executable semantics cannot be executed using the PVS ground evaluator and PVS built-in strategies. For instance, trying to directly execute the filter function in PVSio, which is a PVS evaluator, will release an error message that indicates the filter function includes a non-ground expression.

4.2 Intermediate Type Classification

To respectively generate test cases for OCaml and PVS functions, OPEV needs to catch the commonality between the two languages and dispose of the difference. We, therefore, design an intermediate type system to bridge the gap between the type systems of OCaml and PVS languages. Since the types of the two languages cannot be mapped to each other one-to-one, we divide the types of the two languages into six different classes and make rules to handle them separately.

The six classes of OPEV's intermediate type system, as shown in Listing 4.4, are `PEmpty`, `PBasic`, `PComplex`, `PDef`, `PExt`, and `PSpec`. In the type system, `PEmpty` represents a dummy type, which has no concrete content and is used as a placeholder in the type notation. The other five classes incorporate a subset of the real PVS/OCaml types.

Listing 4.4: Intermediate type classification.

```
type pType =
  | PEmpty
  | PBasic of pBasic
  | PComplex of pComplex
  | PDef of pDef
  | PExt of pExt
  | PSpec of pSpec
```

The relationship between OPEV type classification and OCaml/PVS types is briefly described as follows:

- `PBasic` represents the basic built-in types for both OCaml and PVS, such as `bool`, `nat`, and `int`.
- `PComplex` incorporates the generic complex data types that have similar formats in both OCaml and PVS, such as `string`, `tuple`, and `list`.
- `PDef` illustrates the user-defined types including `datatype`, `record`, and etc.
- `PExt` stands for external library types. These types provide no concrete implementations; instead, they supply specific interfaces that can be used to operate on them. Explicit construction and parsing functions are demanded for these types.
- `PSpec` includes some types that require special treatment such as `functional` types.

Based on the class of each intermediate type, a generating rule and a parsing rule are made. Currently, OPEV manages a subset of the OCaml/PVS type system. To extend the OPEV type system to incorporate new types, one needs to manually add particular test generating rules in OPEV for the new types.

4.3 Test Generation

This section introduces the generating rules for all the intermediate types. Suppose we have a function that is translated from OCaml to PVS. OPEV generates test cases for the function by classifying each of the function arguments into the five intermediate type classes and generating multiple concrete values for the function argument based on its intermediate type representation. Then OPEV normalizes the values to fit them into OCaml and PVS formats.

4.3.1 Basic Types

Types in the `PBasic` class have corresponding built-in types in both OCaml and PVS. The test generating rules are straightforward. As shown in Listing 4.5, the basic types in the intermediate language include `PUnit`, `PBool`, `PChar`, `PNat`, `PInt`, and `PReal`.

Listing 4.5: Basic types.

```
pBasic =
  | PUnit
  | PBool
  | PChar
  | PNat
  | PInt
  | PReal
```

`PUnit` is the intermediate `unit` type for which OPEV will generate () and *Unit* respectively for OCaml and PVS. `PBool` represents the built-in `bool` type. OPEV randomly generates a 0 or 1 and translates it to *false* or *true* for both OCaml and PVS. For `PChar`, OPEV will randomly generate a number between 32 and 126 and then construct a char argument separately for OCaml and PVS according to the type representation.

`PNat` represents the `nat` type that has no explicit definition in OCaml. We set this type based on PVS's type notation. OPEV generates a random natural number in a pre-defined range. Meanwhile, the generating strategy for the `PInt` type is similar to `PNat`. OPEV generates a random integer number in a pre-defined range([-10, 10] by default). The generated integer follows a uniform distribution, and the pre-defined range can be modified by the user in the command line. For instance, if the user needs to change the range to [-5, 5], the corresponding command is as follows:

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

For the `PReal` type, OPEV employs fractional representation. The numerator and denominator are generated as random integer numbers, respectively. The denominator specifically must be nonzero. Then the numerator and denominator are applied to construct the `real` type argument. For OCaml, we have to add the ".0" suffix to them and construct the ratio as *numerator.0/.denominator.0*. Meanwhile, for PVS, the fraction is represented as *numerator/denominator*.

Table 4.1 presents some examples of generated arguments for the basic types. The same arguments can have different representations in OCaml and PVS.

Table 4.1: Examples of generated basic arguments.

Type	OCaml argument	PVS argument
PUnit	()	Unit
PBool	false	false
PChar	'a'	char(97)
PNat	2	2
PInt	-10	-10
PReal	3.0/.5.0	3/5

4.3.2 Complex Data Types

Complex data types in OPEV include `PString`, `PTuple`, and `PList` as shown in Listing 4.6. These types respectively represent the `string`, `tuple`, and `list` types in OCaml and PVS. For `PString` and `PList` types, users can set a length parameter that restricts the maximum length of the elements, and the tests are generated using this parameter. For some recursively defined complex data types, we do not need to deal with the termination issues since these data types have corresponding inherent definitions in OCaml and PVS and OPEV has specific generating rules for each of the built-in types.

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

Listing 4.6: Complex data types.

```
pComplex =
  | PString
  | PList of pType
  | PTuple of pType list
```

A `PString` argument is viewed as a sequence of `char`. OPEV firstly creates a list of random natural numbers, whose ranges are between 32∼126, with a given string length. Then the list of natural numbers is mapped to a list of `char`, and the list of `char` is concatenated to a string based on the normalizing strategies for both OCaml and PVS.

The `PTuple` type has an argument, which is a list of OPEV intermediate types representing the type of each element in the `tuple`. OPEV generates values for each element, based on its type, and combines them together to create the final results. For instance, if the tuple elements are generated as t_0, t_1, ..., and t_n, then the final tuple is (t_0, t_1, ..., t_n).

For a `PList` type, there is also an argument that is the type of the `list` element. To create test cases for a `PList` type, OPEV first generates an integer to represent the length of the list, which is confined to a pre-defined maximum length parameter. Then OPEV generates list elements, following the strategies of creating test cases for the specific list type, with the

Table 4.2: Examples of complex type arguments.

Type	OCaml argument	PVS argument
PString	"\"HelloWorld"	doublequote o "HelloWorld"
PTuple	(2, true, 'a')	(2, true, char(97))
PList	[1; 2; 3]	(: 1, 2, 3 :)

given length. The list elements are concatenated to construct a list for OCaml and PVS, respectively, following their type representations. For instance, if the length of a list is n and the list elements with certain type are generated as x_0, x_1, ..., and x_{n-1}, OPEV constructs an OCaml list as [x_0; x_1; ...; x_{n-1}] and a PVS list as (: x_0, x_1, ..., x_n :).

A few examples of complex data type arguments are shown in Table 4.2. We can see that the double quotation mark within a PVS string is separated out as a constant, called *doublequote*, and combined with the rest string using the `o` infix operator.

4.3.3 User-Defined Types

In OCaml, developers could use the `type` keyword to define a new type that is constructed as a `record` or a `datatype`. The user-defined type is composed of various fields, and each field is denoted with a specific constructor and corresponding type annotation. OPEV sequentially generates test cases for each field of the user-defined type. It should be noted that this generating strategy may cause an infinite loop issue if there exist recursive definitions in the user-defined type. Thus, we set an upperbound for the recursive times to prevent infinite construction.

Moreover, if the return type of a function is a user-defined type, OPEV requires specialized normalizing rules to translate the return results from OCaml to PVS. Namely, if a user needs to employ OPEV to generate tests for a new user-defined type, he/she needs to implement the normalization function in the source code of OPEV.

In the current version of OPEV, the existing user-defined types include `PBit`, `PBvec`, `PRat`, `PSet`, `PRecord`, `PDatatType`, and `PField` as shown in Listing 4.7. The first four types are defined explicitly in the OCaml source in our case studies. `PRecord` and `PDataType` represent built-in type formats in PVS that have generic semantics, though the names and concrete definitions can be different. `PField` is an auxiliary type that is employed to construct the arguments for `PDataType`.

`PBit` and `PBvec` represent `vbit` and `value` types. For the `PBit` type, OPEV randomly generates 1 or 0, constructs either *true* or *false* for PVS, and sets up *Vone* or *Vzero* for OCaml. The `PBvec` type is handled using the following steps: OPEV first creates a list of `PBits` with a given length; then, the list is translated to `string` arguments for OCaml and

Listing 4.7: User-defined types.

```
pDef =
  | PBit
  | PBvec
  | PRat
  | PSet of pType
  | PRecord of (string * pType)
  | PDataType of (string * pType)
  | PField of (string * pType)
```

PVS, respectively, using pre-defined translation functions.

`PRat` represents the `rational` type. In PVS, the `rational` type is a built-in type that is a subtype of the `real` type. To generate test cases for the `rational` type, OPEV employs fractional representation to denote a rational number and generates two random integer numbers respectively as numerator and denominator (denominator cannot be zero). The two integer numbers are then applied to build up `rational` arguments using specific construction functions.

The `PSet` type is defined to stand for the real `set` type. In OCaml, a new `set` type is represented as a `record` consisting of a balanced tree and a comparison function. To handle this `set` type, in OPEV, we implement a construction method to generate a given number of elements according to the specific set member type, and the `set` arguments for OCaml and PVS can be constructed using the construction function respectively.

`PRecord` is the OPEV intermediate notation of the `record` type. A `record` type is similar to a lightweight module or class type that contains multiple named fields representing variables and functions. OPEV does not cover all the types in OCaml. Hence it is challenging to instantiate a `record` type by generating each named field of this `record`. To address the problem, OPEV applies a roundabout strategy. First, OPEV traverses and records all the pre-declared instantiations for the corresponding `record` type in the OCaml source. Then OPEV selects one instantiation based on the outer conditions, and the name of the selected instantiated `record` is used as the final argument.

`PDataType` consists of a name and multiple fields to represent a user-defined `datatype` type. These fields are represented using the `PField` type, which is an auxiliary type for `PDataType`. When OPEV generates test cases for a `datatype`, it looks up a pre-defined hashtable with the name of the `datatype`. Then all the fields' definitions of the `datatype` can be retrieved from the hashtable. Among these fields, OPEV randomly selects one and generates test cases using the constructor and type annotation of the field. The generated test cases for the field also serve as the test cases for the whole `datatype`.

Table 4.3 shows some examples of test arguments for user-defined types. There is no uni-

Table 4.3: Examples of generated user-defined arguments.

Type	OCaml argument	PVS argument
PBit	Vone	true
PBvec	Vvecotr([|Vzero|], 0, true)	LAMBDA (i:below(1)): TABLE | i = 0 | FALSE || ENDTABLE
PRat	Rational.QI.of_ints 3 5	3/5
PSet	Pset.add 2 (Pset.add 1 (Pset.empty compare))	{x: nat| x = 1 OR x = 2}
PRecord	instance_Eq_bool	instance_Eq_bool
PDataType	Some [1; 2]	Some((:1, 2:))

fied format for OCaml and PVS arguments, and each user-defined type requires specific construction functions.

4.3.4 External Types

External types are the OCaml types that are imported from external libraries, which means OPEV does not know the detailed implementations of these types other than the interfaces regarding the types. We have to manually define specific construction functions that map the OPEV intermediate type to corresponding OCaml and PVS types. OPEV does not cover all the external types. For some external types that are used in the libraries in the case studies (Chapter 5), we define generation rules.

For example, in our case studies, a typical external type is `Nat_big_num.num`, which is defined in the library file `nums.cma`. This type is used to handle the situation where big integer computations are carried out. Meanwhile, in PVS, there are no limits on the range of the default `int` and `nat` types. Thus, in PVS, the test cases can be automatically 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.

To represent the external `Nat_big_num.num` type, OPEV introduces `PBigNat` and `PBigInt` types as shown in Listing 4.8.

Listing 4.8: External library types.

```
pExt =
  | PBigNat
  | PBigInt
```

In PVS, `PBigNat` and `PBigInt` types represent the basic `nat` and `int` types, and the test cases

Table 4.4: Examples of generated external arguments.

Type	OCaml argument	PVS argument
PBigNat	Nat_big_num.of_int 5	5
PBigInt	Nat_big_num.of_int (-10)	-10

can be generated following the rules for `PInt` and `PNat`. On the other hand, in OCaml, we introduce a new construction function that turns an integer number into a `Nat_big_num.num` number. This construction function is implemented according to the documentation for the external library.

As shown in Table 4.4, the construction function for both `PBigNat` and `PBigInt` type is `Nat_big_num.of_int`.

4.3.5 Functional Types

The challenge of constructing a functional argument lies in that the function domain and range are potentially infinite. Our strategy of generating concrete test cases for other types is not applicable to the `function` type. We initially considered applying the methods in Haskell QuickCheck [27] 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 behaviorally equivalent functions for OCaml and PVS, we employ a comparatively simple method to generate the functional argument.

First, we define various functions in PVS with specific function patterns. Then OPEV randomly selects a pre-defined function and applies the function name as the PVS argument. Meanwhile, the OCaml argument is the corresponding function name similar to the PVS one.

However, if there are no pre-defined functions for certain patterns or there are no matching PVS and OCaml functions, OPEV constructs a `LAMBDA` expression, which takes symbolic arguments as the inputs and returns a randomly generated value as the output. This `LAMBDA` expression directly serves as the PVS argument, and a corresponding `fun` expression is constructed as the OCaml argument.

As an example, we have a function with pattern $\mathsf{int}- > \mathsf{int}- > \mathsf{int}$, which is represented as $\mathsf{PFunc}[\mathsf{PInt}; \mathsf{PInt}; \mathsf{PInt}]$ in the intermediate type system as shown in Listing 4.9. We have defined a PVS function named `add`, which could be used as the PVS argument. Meanwhile, the corresponding + infix operator can be used as the OCaml argument. However, if none of the functions with this pattern have been defined in the PVS specification, OPEV automatically generates a function using the `LAMBDA` expression.

Listing 4.9: Function argument for specific pattern.

```
Function pattern: [PInt; PInt; PInt]
Pre-defined PVS function: add(x: int)(y: int) : MACRO int = (x + y)
Auto-generated PVS function: LAMBDA (x: int)(y: int): 5
```

4.3.6 Dependent Types

In PVS, a dependent type can be defined explicitly using the `TYPE` keyword or implicitly in a function declaration. The generating strategy is to construct arguments for the dependent type based on its supertype, complying with the constraints of the dependent type. Currently, the supported constraints include arithmetic and comparison operations. Other than these types of constraints, OPEV directly generates 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 restricted by a constant N. OPEV employs the constraint to build up a new range for the natural number and to generate a natural number within the range as a `word` type argument.

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

This test generation strategy does not support more complicated constraints than arithmetic and comparison operations since complicated constraints lead to some redundant test lemmas that OPEV would reject. Although the redundant test lemmas do not cause any inconsistency for the equivalence between OCaml and PVS specifications, they narrow the test coverage for functions with arguments of these dependent types.

4.4 Proof Automation

With the test generation rules, OPEV could automatically generate thousands of test cases for OCaml, thus could construct multiple test lemmas for each of the PVS functions. It is impractical to manually prove all the test lemmas. To automate the proof process, we prove 392 general theorems that provide fundamental properties for many functions, such as the associativity and commutativity of `add` operations for bit-vectors with the same length.

Listing 4.10: 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)
```

Using these general theorems, we implement generic PVS strategies based on the patterns of the functions that are tested. For example, in Listing 4.10, a 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 the combination of addition and negation operation.

We then leverage a utility in PVS called Proof-Lite [71] to prove the test lemmas on the translated functions with the pre-implemented PVS strategies. The strategies will be able to instantiate the general theorems with concrete numbers as in the test lemmas.

Since Proof-Lite validates the test lemmas sequentially, which is not time efficient, we design a `memory management algorithm` to validate the test lemmas concurrently while efficiently managing the memory. In the `memory management algorithm`, OPEV calls multiple processes to validate 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

We implement a set of generic PVS strategies to automatically prove large-scale test lemmas with non-executable semantics in PVS. To construct a generic PVS strategy for different functions, we start from a test lemma and manually prove it. During the manual proof process, we build up a simple PVS strategy for test lemmas with this pattern. Then we attempt to prove other tests with different patterns using this PVS strategy. If this strategy does not work, we prove the new test lemmas manually and get some new PVS strategies. We combine the PVS strategies for test lemmas with different patterns together using techniques such as branching, backtracking, feature extracting, and summarizing. By repeatedly carrying out this procedure, we synthesize a unified pattern behind the validation of the test lemmas. We then build up 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 example, in a basic OCaml-to-PVS translation library named OPEV_Value library (Section 5.1), functions are mainly related to bit-vector operations. The functions in this library involve conversions between natural numbers and their bit-vector representations. The conversion from a natural number to a bit-vector is defined as follows in PVS (the source code is in [84]):

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

The `nat2bv` function is non-executable since it declares that it is the inverse function of `bv2nat`, which defines the conversion from a bit-vector to a natural number. Meanwhile, multiple functions in the OPEV_Value library call this `nat2bv` function. Thus, to prove test lemmas containing nat2bv function, which has non-executable semantics, we exploit the relation between nat2bv and bv2nat functions to circumvent the execution of nat2bv function.

For example, the case-split-strat strategy, as illustrated in Listing 4.11, employs the injectivity and invariance properties of the nat2bv and bv2nat functions. This PVS strategy is extensively used to prove test lemmas for functions in the OPEV_Value library.

Listing 4.11: 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 completing the generic PVS strategy, we employ 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. With the PVS strategy, OPEV is capable of efficiently validating hundreds of thousands of test lemmas automatically. The statistics are illustrated in Chapter 5.

Chapter 5

Case Studies of OPEV

We 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 detect 11 mismatches during the validation of these case studies. Documentation on these bugs is available in [76]. The validation is carried out on an AMD Opteron server (2.3GHz, 64 core, 128GB).

5.1 Manually Implemented OCaml-to-PVS Translation

OPEV validates a manually implemented PVS library for which the source is a single OCaml file in the Sail source code [93], 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, and it is necessary to increase the reliability of the translation. Table 5.1 illustrates the statistics for this validation.

We validate ∼200K test lemmas and find six mismatches. For example, a function named add_overflow_vec_bit_signed carries out two's complement bit-vector addition operation. In the implementation of the function in PVS, if the second operand is negative, we ascertain that there is no overflow and no carry bit for the addition operation. However, in one version of sail_values.ml [93] (commit ce962ff), overflow is set to true. Thus, there exists a conflict in the two implementations, and the results parsed from the execution of the OCaml function cannot be validated in the PVS test lemmas. OPEV reports 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 Mismatches Found	6

5.2 Sail-to-PVS Parser

The Sail language [43], which is a first-order imperative language, has been employed to describe the semantics of ISAs such as x86, ARM, RISC-V, and PowerPC [43]. To facilitate the reasoning on these semantics, we implement 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 [93] to automatically translate Sail source code to Lem [64], which is designed to serve as a semantic model that is mathematically rigorous [100] 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 [64].)

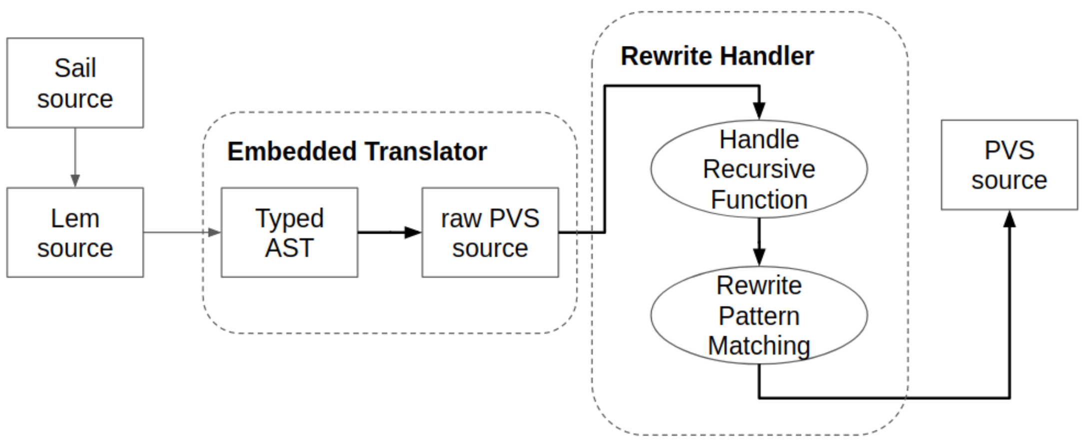

Figure 5.1: Architecture of the Sail-to-PVS parser.

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 Lem source and translates the typed AST into corresponding PVS code using PVS syntax. This step is challenging since the Lem type system does not support dependent types, which are widely used in PVS. Besides, Lem originally was designed to translate Sail specifications into theorem proving languages that do not support dependent types, such as HOL4 and Isabelle. In addition, at this stage, the generated raw PVS code is error-prone due to 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 are different in PVS and Lem.

We apply a rewrite handler written in Python to adjust the problematic PVS code. The rewrite handler performs two tasks: rewriting the pattern matching to ensure that the PVS code has consistent types and adding `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. Meanwhile, the Sail-to-PVS parser is still restricted to pure functions in Sail with these modifications.

An important usage of the Sail-to-PVS parser is program verification at the assembly level (using PVS). For such a usage, it is critically important that the translation is provably correct. We automatically translate a Lem basic library [64] 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; meanwhile, all of them are pure functions. 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 source using OPEV. If the equivalence is validated, our trust that the Sail-to-PVS parser carries out similar functionality as the Sail's built-in compiler will increase significantly. Thus, the Sail-to-PVS parser is reliable if the Sail's built-in compiler is trustworthy.

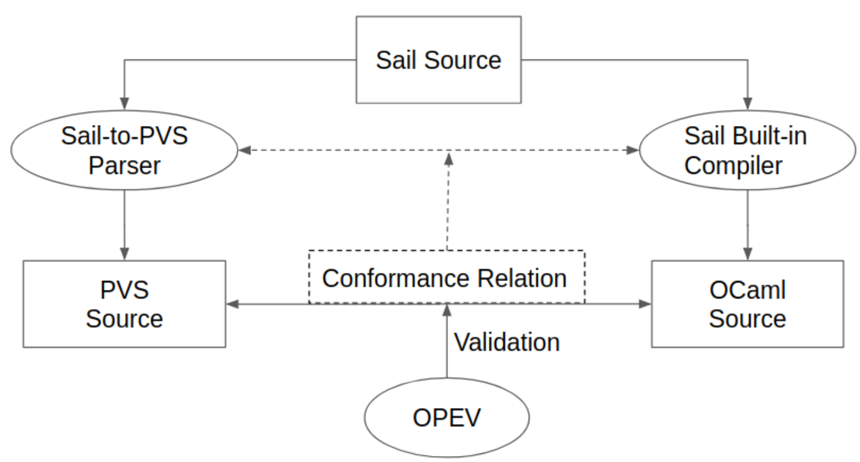

Figure 5.2: Application of the OPEV methodology to validate the Sail-to-PVS parser.

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

Table 5.2 shows the statistics for the library. OPEV determines multiple unprovable test lemmas in the PVS implementation. In turn, we modify the source code of the Sail-to-PVS parser, which generates the test lemmas reported in the table. Due to the gap between the

Table 5.2: Statistics on validation of the 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

semantics of the Lem and PVS languages, OPEV detects five mismatches. Without OPEV, it is practically impossible to manually complete the validation on the translation.

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 a 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 to.

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 validate the soundness of these models. However, machine instructions are written in a binary file with byte sequences. The formal validation of the disassembly 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 unconvinced since it cannot cover all the possible paths during the execution.

In this chapter, we provide a soundness definition of a disassembly process in Section 6.1. Moreover, in Section 6.2, we discuss a critical assumption required to ensure that the soundness definition reflects the correctness of a disassembly process without ground truth.

6.1 Soundness Definition

To formulate a formal notion of disassembly soundness, we first introduce the types and notations used in the definition. An element of type `Nword` is a bit vector with size `N`. Given a bit vector w, notation $|w|$ provides the size of the bit vector. The type `Instruction` indicates the type of valid x86-64 instructions. In our soundness definition, an instruction is represented by, among other things, an opcode mnemonic, its operands with size directives, and possibly certain prefixes.

The definition of soundness is based on three essential components: a function `read_bytes` that reads byte sequence from a binary file, a function `bytes_of` that assembles a single instruction into bytes, and an abstract transition relation $\rightarrow_A$.

The first function read_bytes reads, given an address and a size, a byte sequence from the binary file. In all the following definitions, the type of the address is expressed as 64word, and the type of byte is 8word. Then the type annotation of read_bytes is:

$$\mathsf{read\_bytes} : \mathtt{64word} \mapsto \mathbb{N} \mapsto [\mathtt{8word}]$$

Function bytes_of maps a single instruction to the corresponding byte sequence representation, which is the basic work of any assembler. Although the bytes_of function represents an assembly process, our soundness definition does not consider any specific implementation of an assembler. Function bytes_of is type-annotated as:

$$\mathsf{bytes\_of} : \mathtt{Instruction} \mapsto [\mathtt{8word}]$$

Let $\rightarrow_C$ denote a deterministic concrete transition relation over concrete addresses, and $\rightarrow_C^*$ represents the transitive closure of this transition relation. Modeling this concrete transition relation is impossible: the relation depends on the current state of registers, memory, and flags, but also on the state of peripherals, the OS, etc. Let a_0 be a binary's entry address. An instruction address a is *reachable* at run-time, if and only if:

$$a_0 \rightarrow_C^* a$$

The soundness definition is based on an over-approximative abstraction of this concrete transition relation, which is defined as $\rightarrow_A$. This is a non-deterministic transition relation over addresses: $\rightarrow_A$ is of type $\mathtt{64word} \mapsto \{\mathtt{64word}\}$. This transition relation solely concerns the 64-bit value of the instruction pointer rip of the concrete state and produces a set of next instruction addresses.

Definition 6.1. Transition relation $\rightarrow_A$ *is a proper abstraction of* concrete transition relation $\rightarrow_C$, if and only if, for any reachable concrete states s and s':

$$s \rightarrow_C s' \implies \mathtt{rip}(s) \rightarrow_A \mathtt{rip}(s')$$

We use $\rightarrow_A^*$ to indicate the transitive closure of $\rightarrow_A$.

Finally, the input of the soundness definition is the output of a disassembly process. This output basically is a partial mapping from byte sequence to instructions. It is denoted as disasm. We also define an auxiliary function disasm_n. Function disasm_n returns, with the given current address, the size of the byte sequence that is to be disassembled for the next single instruction. The two functions are of type:

$$\mathsf{disasm} : [\mathtt{8word}] \mapsto \mathtt{Instruction}$$

$$\mathsf{disasm\_n} : \mathtt{64word} \mapsto \mathbb{N}$$

Definition 6.2. Let a_0 be a binary's entry address and let disasm be some disassemblers' output. Output disasm is *sound*, if and only if:

$$\forall a \cdot a_0 \rightarrow_A^* a \implies \mathsf{bytes\_of}(\mathsf{disasm}(\beta)) = \beta$$
$$\textbf{where} \quad \beta = \mathsf{read\_bytes}(a, \mathsf{disasm\_n}(a))$$

Definition 6.2 indicates that for all reachable addresses a inside a binary file, the bytes β of the disassembled instruction $\mathsf{disasm}(\beta)$ located at address a are equal to the actual bytes that are read from the binary. If there exist some reachable instructions whose bytes are not equal to those in the binary, the disassembler is unsound.

This definition is independent of the inner mechanism of a disassembler. Whether a disassembler is implemented using recursive traversal, linear sweep, or machine-learning is irrelevant since we only try to validate the consistency between a binary file and the output of the disassembler. We treat a disassembler as a black box and only consider the output.

6.2 Loose Comparison of Instruction Bytes

For each reachable instruction address, Definition 6.2 compares the bytes produced by reassembling a disassembled instruction with the original bytes from the binary. However, a strict byte-to-byte comparison may incorrectly classify a disassembly process as unsound. Consider Figure 6.1. The original assembly process is modeled as a asm function, which maps an instruction to the corresponding bytes. This function is part of the trust base, and it is not available.

$$\mathsf{asm} : \texttt{Instruction} \mapsto [\texttt{8word}]$$

The ground truth is the original instruction i_0, assembled by the original assembler asm to b_0. Both i_0 and asm are assumed to be unavailable. The black-box disassembler disasm produces an instruction i_1 from b_0. Definition 6.2 suggests that it suffices to reassemble instruction i_1 into bytes b_1 and then strictly compare b_0 and b_1 to validate the soundness.

This, however, is not necessarily correct for two reasons. First, the function disasm may produce an instruction different from i_0 but with the same semantics. In such a case, reassembling may not reproduce the same bytes. Second, function $\mathsf{bytes\_of}$ may be different from the original assembler asm (since the original assembler is unavailable). Thus, even if the disassembler under investigation disasm was able to reproduce the exact instruction i_0, a strict comparison between b_0 and b_1 may still fail in the soundness validation.

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

```
objdump(0f1f440000) = nop DWORD PTR [rax+rax*1+0x0]
gcc(nop DWORD PTR [rax+rax*1+0x0]) = 0f 1f 04 00
objdump(0f1f0400) = nop DWORD PTR [rax+rax*1]
```

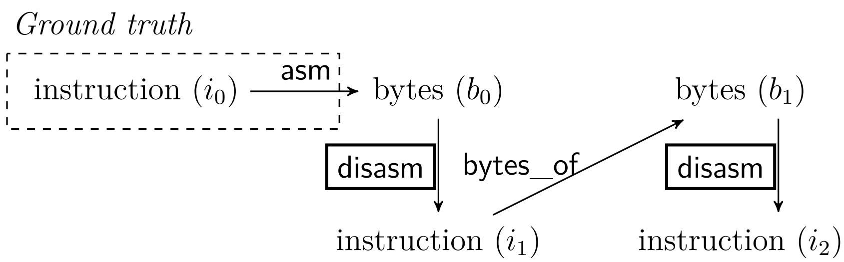

Figure 6.1: Comparison per instruction. The dashed box indicates that the ground truth, i.e., the original instruction and original assembler, are unavailable. The disassembler under investigation (disasm) is black-box.

For example, we employ gcc as the assembler and objdump as the disassembler and get the example in Listing 6.1. In this example, b_0 is `0f 1f 44 00 00`, b_1 is `0f 1f 04 00`. They are not equivalent. If we solely compare b_0 and b_1, we will 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 semantically equivalent. The reason behind this situation is that gcc would automatically carry out optimization when it encounters certain types of instructions.

Thus, instead of a strict comparison, we will use a loose comparison of bytes. The bytes b_1 produced by reassembling are again disassembled. This produces instruction i_2. We will consider b_0 and b_1 loosely equal if these instructions are equal *after normalization*. The *normalization* is executed by a normalize function, which rewrites an instruction to a normalized format following rules such as re-formatting assembly code from AT&T format to Intel, removing `*1` and `+0`, and normalizing the representation of memory accesses. The normalized instruction is ensured to be semantically equivalent to the original instruction.

Definition 6.3. Let β_0 and β_1 be two byte sequences. They are *loosely equivalent*, notation $\beta_0 \simeq \beta_1$, if and only if:

$$\begin{array}{rcl} & \multicolumn{2}{l}{\beta_0 = \beta_1 \vee \mathsf{normalize}(i_0) = \mathsf{normalize}(i_1)} \\ \textbf{where} \quad i_0 & := & \mathsf{disasm}(\beta_0), \\ i_1 & := & \mathsf{disasm}(\beta_1) \end{array}$$

We can now summarise a fundamental part of the TCB of our approach. Since there is no ground truth, this must be assumed and cannot be proven.

Assumption 1. For any instruction i_0:

$$\mathsf{asm}(i_0) \simeq \mathsf{bytes\_of}(\mathsf{disasm}(\mathsf{asm}(i_0)))$$

implies that instruction i_0 has been correctly disassembled by function disasm.

Chapter 7

DSV: Validation Algorithm

In Chapter 6, we define the soundness of the output of a disassembler w.r.t. the original binary file. According to that definition, there are three components that must be implemented: read_bytes, bytes_of, and the abstract step function $\to_A$.

The first two are straightforward. For read_bytes, we employ the `readelf` utility to get the binary segment information and implement a Python program to read a byte sequence from a binary file directly. To implement function bytes_of, we need to translate *a single instruction* to its byte-sequence representation. The choice of the assembler, whether gcc, clang, or some other, is independent of the disassembler under investigation and of the type of the source binary file.

The third component, an abstract transition relation $\to_A$, is more involved. A perfect and exact implementation of this component does not exist since it is undecidable which addresses are reachable from the entry point [87]. It is also undecidable to distinguish instructions from raw data [115]. Implementation of $\to_A$ requires, among other things, dealing with indirect jumps and calls, jump tables, data inlined in code, and overlapping instructions. Specifically, predicting where an indirect branch jumps to is a major challenge for all existing disassemblers.

In this chapter, we introduce the definition of an inexact abstract transition relation and the corresponding consequences in Section 7.1. Then we provide an overview of DSV in Section 7.2. We present the state and memory model of a computer system in Section 7.3. A solution for the state explosion problem is explained in Section 7.4. Section 7.5 introduces how DSV builds up instruction semantics for X86/64 ISA. In Section 7.6, we briefly illustrate some challenges and the corresponding solutions we implement in DSV.

7.1 Consequences of An Inexact Abstract Transition Relation

We introduce an *inexact* abstract transition relation since an exact implementation of the abstract transition relation $\to_A$ is not feasible. We will use $\leadsto_A$ to denote this inexact implementation of the hypothetical exact abstract transition relation $\to_A$. We introduce the following terminology (here a_0 denotes the binaries' entry point):

White An instruction address a is white if it is deemed reachable by the implementation $\rightsquigarrow_A$, i.e.:

$$a_0 \rightsquigarrow_A^* a$$

We can now rephrase the notions of false positive and false negative w.r.t. this terminology. A *false positive* occurs when disassembler-output is deemed sound by DSV, whereas it is incorrect. We define a false positive as the existence of an incorrectly disassembled reachable instruction that is not white. It is thus reachable at runtime and deemed unreachable (and therefore missed) by the implementation $\rightsquigarrow_A$. A *false negative*, then, is an incorrectly disassembled unreachable instruction that is white. In other words, it is deemed reachable by the implementation $\rightsquigarrow_A$, but unreachable at runtime.

A false positive can happen if the implementation $\rightsquigarrow_A$ *under-approximates* the concrete transition relation $\rightarrow_C$. In other words, it can happen if it is possible that a reachable instruction is not white. We aim for an implementation that does not suffer from false positives and therefore require the implementation to be proper (see Definition 6.1): any reachable instruction is visited. In the case of proper over-approximation, a false negative can happen, i.e., an unreachable instruction may be white.

Finally, we would like to note that there is no decidable way to determine whether an instruction address is reachable or not. There is no ground truth and no reliable way of establishing reachability without source code. In practice, however, it is possible to establish the unreachability of certain parts of the binary. For example, in the current implementation, functions called inside an external `__cxa_atexit` function are not considered to be reachable (e.g., deconstructors). We thus use the following terminology:

Black An instruction address is black if it is not white and *it can be established* (e.g., with conservative manual inspection) that it is unreachable.

Grey An instruction address is grey if it is not white and it is not black, i.e., if it cannot be established whether it is reachable or not.

Given an over-approximative implementation $\rightsquigarrow_A$, all instruction addresses reported by some disassembler are either white, black, or grey. The aim is to construct an implementation $\rightsquigarrow_A$ that minimizes the number of grey instructions. Only the case where DSV finds an issue in a grey instruction constitutes a false negative.

7.2 DSV Overview

In essence, DSV employs a standard forward BMC exploration loop. At all times, three parameters are maintained:

s**: the current state.** A symbolic state is maintained that contains symbolic expressions for registers, flags, and memory. The initial state solely consists of an assignment of some concrete values to the stack pointer `rsp` and the instruction pointer `rip`.

π**: the current path constraint.** A symbolic predicate is maintained that contains the branching conditions of the current path. Its purpose is to prune inconsistent paths (we check the consistency using the Z3 SMT Solver [32]). Initially, this constraint is true.

Σ**: the stored states.** A key-value mapping with as keys instruction addresses and as values symbolic states. This mapping allows DSV to keep track of which addresses have been visited and to reduce the traversed state space. Initially, this mapping is empty.

DSV first fetches the instruction i as disassembled by the disassembler under investigation and validates that instruction (see Section 6.2). It then updates Σ by adding the current state σ. It may be the case that the current instruction address was already visited. In that case, a *merge* must happen between the current state s and the stored state. If the current state s and the merged state *agree* (intuitively: they contain the same information), then no further exploration is necessary. If the instruction address was unvisited, the current state is inserted into Σ. DSV then concolically executes instruction i to the merged state s_m, given the current path constraint π. This provides a set of pairs of symbolic states and path constraints; one instruction may induce multiple paths. Each of these pairs is explored.

7.3 State and Memory Model

The state consists of assignments of symbolic expressions to flags, registers, and memory. Symbolic expressions consist of expressions with a standard set of operators (e.g., $+$, $-$, $\ldots$) and as base operands either immediate values, registers, or flags. Most notably, a symbolic dereference operator is supported that reads data from memory. An operand may also be an unconstrained, universally quantified variable. We will use v_f to denote a fresh variable. The symbolic expressions used by DSV are close to that used in existing literature [16].

Since the bit length of all registers is fixed, we model general-purpose registers as a 64-bit Z3 bit-vector and deal with register aliasing accordingly. We set the initial values of all the registers, except for `rip` and `rsp`, to symbolic values and modify the values of registers according to the semantics of instructions. The value of each register can be either symbolic or concrete.

There are different techniques to model memory. To design a space-efficient memory model that simulates the memory changes during the execution of a binary, we model memory as a function mem of type $\texttt{64word} \mapsto ([\texttt{8word}], \mathbb{N})$. This function maps memory addresses to byte sequences and the size of the region starting at the given address. Function mem is partial, which means that not all addresses at the memory have explicit content. At all times, all regions in the range of mem are separate.

In concolic execution, a memory address is either symbolic or concrete. Reading from or writing to a concrete address follows specific rules. For example, as illustrated in Figure 7.1, before the execution of `mov QWORD PTR [1000], 0xaaf1343`, there are two addresses in the domain of function mem: 998 and 1004. Thus we have $\mathsf{mem}(998) = (0x10002, 4)$ and $\mathsf{mem}(1004) = (0x0012, 6)$. Now if we need to read the memory at address 1000 with size 2, we get $0x1$ as the result, which is splitted from $\mathsf{mem}(998)$. After the execution of `mov QWORD PTR [1000], 0xaaf1343`, we write 0xaaf1343 with size 8 to address 1000. This will affect the results 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 $\mathsf{mem}(998) = (0x2, 2)$, $\mathsf{mem}(1000) = (0xaaf1343, 8)$, and $\mathsf{mem}(1008) = (0x0, 2)$.

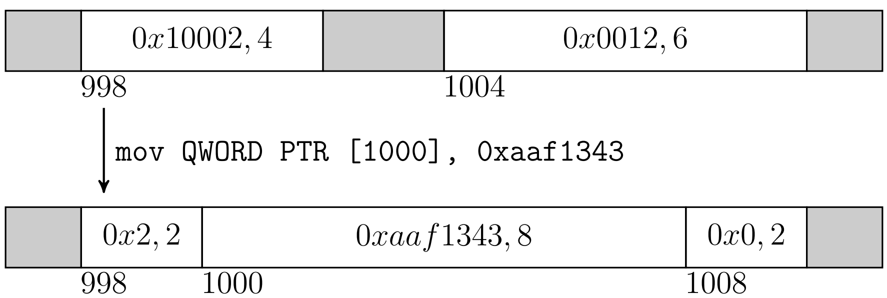

Figure 7.1: An example of a memory writing operation on the memory model.

Since we keep the stack pointer concrete, all local variables correspond to memory regions with concrete addresses. The same holds for global variables. Moreover, the Glibc functions malloc and calloc are modeled in such a way that they return a *concrete* address that does not overlap with any existing region in the memory. This concretizes the majority of addresses. Theoretically, this approach may lead to unsoundness issues. For example, if a program successfully allocates memory using malloc, then branches are taken based on whether that (non-null) pointer is greater than some immediate value. To the best of our knowledge, such behavior is undefined according to the C standard.

Assumption 2. We assume that the control flow of a binary does not depend on the concrete values returned by memory allocation functions or on the concrete value of the stack pointer.

However, not all memory addresses are concrete: symbolic addresses occur when pointers are returned by external functions that are not linked statically. In these cases, reading from a symbolic memory region returns a fresh symbol. Writing to such a memory region will remove all heap-related regions from the memory but will keep the local stack frame intact.

7.4 Merging and Agreeing

If the address of the current state s was already visited, the current state s and the visited state s_{old} are merged (see Algorithm 1). If the current value v at a key k in s is symbolic, then v is possible to represent any value, and we do not need to change it. However, if the current value v is concrete, we need to compare v with v_{old} at the same key k in s_{old} to decide how to merge v and v_{old} to get the new result.

Algorithm 1 Merging algorithm.

function MERGE(s_{old}, s)
    $s_{new} \leftarrow copy(s)$
    for all $(k, v) \in s$ **do**
        $v_{old} \leftarrow s_{old}[k]$
        if v is a concrete value **then**
            if v_{old} is a concrete value **then**
                if $v \neq v_{old}$ **then**
                    $s_{new}[k] \leftarrow$ fresh variable
                end if
            else
                $s_{new}[k] \leftarrow$ fresh variable
            end if
        end if
    end for
    return s_{new}
end function

The current state s is not explored if state s and merged state s_m contain the same information, i.e., if the two state *agree*. Two states agree if they have the same keys and for any key-value pair (k, e) in s and (k, e_m) in s_m the expression e and e_m agree.

Definition 7.1. Let $\mathsf{fresh}(e)$ denote the set of fresh variables in symbolic expression e. Two expressions e_0 and e_1 *agree* if and only if there exists a bijection β between $\mathsf{fresh}(e_0)$ and $\mathsf{fresh}(e_1)$, such that e_0 and e_1 are syntactically equal if all fresh variables v_f in e_0 are replaced with $\beta(v_f)$.

Example 7.2. Consider a loop in which register `rax` is incremented with 4 every iteration. Let the visited state $s_{old} = \{\texttt{rax} := v_{f_0}, \texttt{rdi} := v_{f_0} + 100\}$. After one loop iteration, the current state $s = \{\texttt{rax} := v_{f_0} + 4, \texttt{rdi} := v_{f_0} + 100\}$. The merged state will be $s_m = \{\texttt{rax} := v_{f_1}, \texttt{rdi} := v_{f_0} + 100\}$ and will be stored. States s_m and s do not agree and exploration will continue. However, after one more iteration, we will obtain state $s' = \{\texttt{rax} := v_{f_1} + 4, \texttt{rdi} := v_{f_0} + 100\}$. States s' and state s_m will be merged, resulting in $s'_m = \{\texttt{rax} := v_{f_2}, \texttt{rdi} := v_{f_0} + 100\}$. States s_m and s'_m do agree, and therefore the loop is not unrolled further.

7.5 Instruction 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. We only need to build up semantics for instructions that – be it directly or indirectly – influence the `rip` register. We will call this the set of *relevant* instructions.

The set of *relevant* instructions include `push`, `pop`, `mov`, `lea`, `call`, `ret`, simple arithmetic instructions, logical instructions, bitwise instructions, jump instructions, etc. According to the statistics taken in some literature [3], these instructions would make up over 96% of instructions in multiple C/C++ applications and web browsers. Advanced instructions such as floating-point instructions and SIMD extensions typically do not impact register `rip`. It is not necessary to construct specific semantics for these instructions.

For all the irrelevant instructions, we use *unknown* semantics by assigning fresh variables any time an irrelevant instruction is executed. In most cases, an instruction has an opcode and different operands, and the content of the destination operand is modified by the instruction. For irrelevant instructions, the semantics assigns some fresh variable v_f to the destination operand, representing that the current status of the corresponding register, flag, or memory is undefined or undetermined. The fresh variables are handled using the symbolic execution rules in our DSV SE engine.

7.6 Concolic Execution

As discussed in Section 7.3, we make use of concolic execution that concretizes memory addresses as much as possible while leaving the remainder as symbolic as possible. As such, the branching conditions that are taken are generally symbolic. In the case of a conditional jump based on a symbolic flag value, both paths are taken (sequential execute and jump). This over-approximates reachability.

A key challenge is to resolve indirect-branch addresses. An indirect branch is a control flow transfer (jump or call) where the target is computed instead of an immediate. Indirect branches happen, e.g., in the case of compiled switch statements, function callbacks, or virtual tables. Three cases may arise:

1. The current state is sufficiently concrete that the computation can be resolved. In this case, exploration continues.

2. The expression that computes the next value of `rip` is symbolic; however, the current state and the path constraint contain sufficient information to both bind and over-approximate the set of next addresses. In this case, exploration continues to all next addresses.

3. The current state does not contain sufficient information to bind the set of next addresses; the expression that computes `rip` contains unbounded symbolic values. An error message is produced, and we manually investigate how to resolve the issue. Generally, we need to trace back and see which irrelevant instructions should be considered relevant. This situation is infrequent since we have modeled the semantics of the most common instructions based on their usage rate.

With the state model for registers, flags, and memory, we carry out the concolic execution to construct a CFG for the machine code. Concolic execution is over-approximative. The vast majority of branches are taken due to symbolic conditions. Meanwhile, `rsp` is always concrete, and therefore local variables in the stack frame can be read/written. Besides, addresses are concrete in the memory allocation functions. The concrete addresses prevent memory aliasing issues.

In the construction of CFG, indirect jump, indirect call, and return instructions pose a challenge in how to resolve the indirect-branch addresses. The path constraint provides a bound on the set of next addresses. Besides, we introduce a trace-back model to fix the problem of unimplemented instruction semantics. We also implement an algorithm [23] to solve the challenge of jump table without determined upperbound. However, there still exists unresolved indirect-branch addresses in the concolic execution since it is an undecidable problem.

Chapter 8

DSV: Experimental Results

After we implement DSV that realizes the soundness definition of a disassembly process, we employ the tool to validate the disassembly carried out on the Coreutils library using different disassemblers. Section 8.1 introduces some of the soundness issues which are detected by DSV. In Section 8.2, we apply DSV on eight different disassemblers: objdump 2.30, radare2 3.7.1, angr 8.19.7.25, BAP 1.6.0, Hopper 4.7.3, IDA Pro 7.6, Ghidra 9.0.4, and Dyninst 10.2.1, using 102 test cases from Coreutils-8.31. Here, we evaluate the performance of DSV.

All these experiments are carried out on a machine with Intel Core i7-7500U CPU @ 2.70GHz $\times$ 4 and 16GB RAM. The OS is Ubuntu 20.04.2 LTS, and the Coreutils-8.31 library is compiled using gcc 7.5.0 through the standard build process.

8.1 Soundness Issues Exposed by DSV

This section summarises some of the soundness issues found by DSV. We mainly focus on instructions that are erroneously recovered by different disassemblers.

In Section 8.2, we use DSV to evaluate the disassembly results generated by eight disassemblers on the Coreutils library. Even though most of the reachable instructions for these disassemblers are correctly recovered, there are few exceptions where the disassembled instruction is incorrect w.r.t. the byte sequence. We report on some cases found by DSV that are inappropriately disassembled by certain disassemblers. Table 8.1 summarises the found results, which are disagreed for different disassemblers. Some of the disagreements (row 1, 2 of the table) are trivial and can be argued not to impact soundness. Row 3, 4, 5, and 6 of the table consist of actual soundness issues.

Row 1 and 2 of Table 8.1 mainly concern different representations of the same semantical intent. There are cases where the operands of an instruction are not represented since default behavior is assumed. For instance, both Ghidra and Dyninst (correctly) assume that immediates are sign-extended to fit the destination operand, if necessary. However, minor differences may be relevant. For example, the instructions `repz ret` and `ret` have the same semantical intent but their execution time may differ for certain architectures.

Row 3, 4, 5, and 6 concern semantically different instructions. For instance, Dyninst disassembles `4899` to `cdq rax`, which is not a valid instruction in x86-64 ISA (note that `cdq`

Table 8.1: Examples of instruction recovery results for different disassemblers. All the results are normalized to Intel format.

<table>
<tr><th>bytes</th><th>objdump</th><th>radare2</th><th>angr</th><th>Hopper</th><th>BAP</th><th>IDA Pro</th><th>Ghidra</th><th>Dyninst</th></tr>
<tr><td>f3c3</td><td>repz ret</td><td colspan="3">ret</td><td>rep ret</td><td>rep retn</td><td>ret</td><td>rep ret</td></tr>
<tr><td>4881a4249000 0000fffbffff</td><td colspan="6">and qword ptr [rsp+0x90], 0xfffffffffffffbff</td><td colspan="2">and qword ptr [rsp+ 0x90],0xfffffbff</td></tr>
<tr><td>4899</td><td colspan="7">cqo</td><td>cdq rax</td></tr>
<tr><td>4d0fa3f7</td><td colspan="6">bt r15,r14</td><td>bt rdi ,r14</td><td>bt r15,r14</td></tr>
<tr><td>48be00000000 00f0ffff</td><td colspan="5">movabs rsi, 0xfffff00000000000</td><td colspan="2">mov rsi,0xffff f00000000000</td><td>mov rsi,0x-17 592186044416</td></tr>
<tr><td>64488b042528 000000</td><td colspan="7">mov rax,qword ptr fs:[0x28]</td><td>mov rax,0x28</td></tr>
</table>

performs sign-extension to 64 bits, whereas `cqo` performs sign-extension to 128 bits). An example is shown where Ghidra misrepresents a register (`rdi` instead of `r15`). Besides, a 64-bit immediate is wrongly disassembled by Dyninst. Finally, Dyninst sometimes seems to omit representations of segment registers such as `ds` and `fs`.

Except for the examples listed in Table 8.1, there are some ambiguous cases for different disassemblers. The outputs generated by Dyninst do not have any `ptr` operator to indicate the operand size of a memory operand, which leads to ambiguous semantical behavior. For example, `49837c242800` is translated to `cmp [r12 + 0x28],0x0` by Dyninst while the other disassemblers' result is `cmp qword ptr [r12+0x28],0x0`. Without the `qword ptr` specifying the size of the operand as 64-bit, we cannot determine what the exact value reading from the memory is. Thus the result of the `cmp` instruction is undetermined.

8.2 Coreutils Library

We apply DSV on 102 test cases in the Coreutils library, which are disassembled using eight disassemblers. For each test case, we report the number of instructions: total, white, gray, and black. The definition of *white*, *black*, or *grey* instructions are given in Section 7.1. Roughly speaking, white indicates instructions that are proven to be reachable by DSV, and black illustrates unreachable instructions. The grey instructions are those that are reported by the disassembler but are not visited by DSV; the reachability of these instructions is unknown. Table 8.2 shows the results of `basename`, `expand`, `mknod`, `realpath`, and `dir` test cases in the Coreutils library for different disassemblers. These 5 test cases are selected based on the number of total instructions and the diversity of various instruction types.

Table 8.2: Execution results for Coreutils library on different disassemblers. Only 5 of 102 binaries are shown.

		# of total	# of white	# of grey	# of black	Ratio of grey vs. white	# of indirects	Missing instr	Sound
objdump	basename	3310	2217	18	1075	0.01	59		
	expand	3928	2742	112	1074	0.04	79		
	mknod	4101	2775	216	1110	0.08	65		
	realpath	5828	2644	89	3095	0.03	72		
	dir	19029	12751	417	5861	0.03	230		
radare2	basename	3409	2217	18	1174	0.01	59		
	expand	4027	2742	111	1174	0.04	79		
	mknod	4200	2775	214	1211	0.08	65		
	realpath	5927	2644	86	3197	0.03	72		
	dir	19124	12900	320	5904	0.02	231	×	×
angr	basename	3415	2217	18	1180	0.01	59		
	expand	4033	2742	111	1180	0.04	79		
	mknod	4206	2775	214	1217	0.08	65		
	realpath	5933	2644	86	3203	0.03	72		
	dir	19134	12751	413	5970	0.03	230		
BAP	basename	5894	826	114	4954	0.14	37	×	
	expand	7373	1320	205	5848	0.16	56	×	
	mknod	7022	1282	162	5578	0.13	43	×	
	realpath	11368	1251	108	10009	0.09	46	×	
	dir	28906	5718	667	22521	0.12	150	×	×
Hopper	basename	3250	2217	18	1015	0.01	59		
	expand	3845	2742	111	992	0.04	79		
	mknod	4022	2775	68	1179	0.02	65		
	realpath	5636	2644	86	2906	0.03	72		
	dir	18292	12607	350	5335	0.03	230	×	×
IDA Pro	basename	3221	2217	18	986	0.01	59		
	expand	3820	2742	111	967	0.04	79		
	mknod	3995	2775	68	1152	0.02	65		
	realpath	5607	2644	87	2876	0.03	72		
	dir	18220	12751	268	5201	0.02	230		
Ghidra	basename	3256	2217	18	1021	0.01	59		
	expand	3826	2742	99	985	0.04	79		
	mknod	4029	2775	68	1186	0.02	65		
	realpath	5658	2644	86	2928	0.03	72		
	dir	18303	12751	267	5285	0.02	230		×
Dyninst	basename	3269	2222	16	1031	0.01	60		×
	expand	3874	2707	123	1044	0.05	79		×
	mknod	4058	2747	214	1097	0.08	64		×
	realpath	5724	2609	85	3030	0.03	71		×
	dir	18694	12845	329	5520	0.03	230		×

8.2.1 Instruction Recovery

Most disassemblers are capable to correctly disassemble all the reachable instructions. As shown in Figure 8.1, for most of test cases in Coreutils library, objdump, angr, BAP, and IDA Pro achieve an accuracy rate of 100% for single-instruction recovery. Meanwhile, Ghidra and Dyninst make some errors in the disassembly process for some test cases, and the accuracy would decrease to around 97.5%.

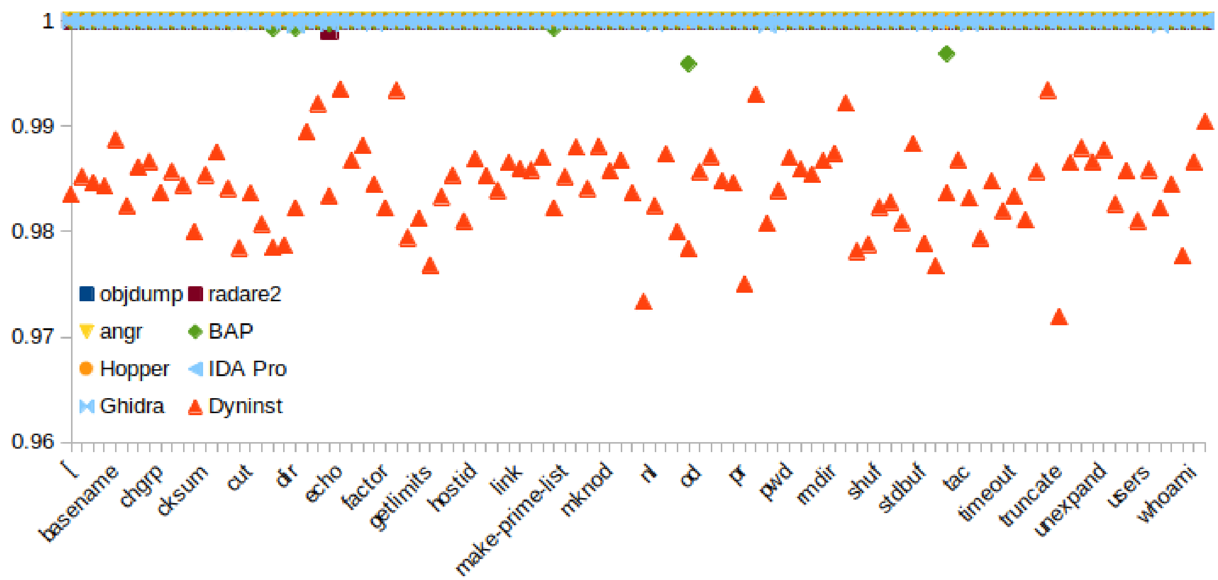

Figure 8.1: Ratio of correctly disassembled vs. the white disassembled instructions.

8.2.2 Control Flow Recovery

For all test cases, there exists a gap between the number of white instructions, which are reachable instructions detected by DSV, and the number of total instructions; in other words, the number of black instructions can be relatively high. This can be accounted for two reasons.

The first reason is that different disassemblers consider different parts of the binary. For example, BAP generates the instructions from sections .symtab, .debug_line, .debug_ranges, and so on, while some disassemblers may solely generate instructions from .text, .plt, and .plt.got sections.

The second reason lies in the technique that DSV employs to handle external functions. DSV treats external functions as black boxes and does not go inside the external functions to execute them. Internal functions that are called by external functions may be considered black. For example, the internal function `close_stdout` is called by the external function `___cxa_atexit` (it calls the close function after program exit). Thus, the `close_stdout` function is considered black. Some exceptions include `__libc_start_main` and `pthread_create`. These two external functions execute the function pointer passed through the `rdi` register, and the internal functions pointed to are not executed by DSV. Broader coverage, i.e., fewer black instructions, can be reached by providing semantics to external functions that call internal ones.

The ratio of grey vs. white instruction is an indication of how accurate control flow has been recovered. If the ratio is low (zero), then the disassembler highly accurately decides which instructions are reachable and which are not. If it becomes higher, this may indicate either the disassembler coarsely over-approximated which instructions are reachable (many grey instructions) or the disassembler missed instructions. The ratio is, on average, about 4%. As shown in Figure 8.2, BAP usually has the highest ratio since the instructions whose addresses are stored in indirect jump tables are missed by BAP due to lack of support for indirect branching. Meanwhile, objdump and angr have similar ratio for most of test cases , as we use angr to statically generate a CFG (CFGFast) and to disassemble a binary file, which have similar outputs as objdump.

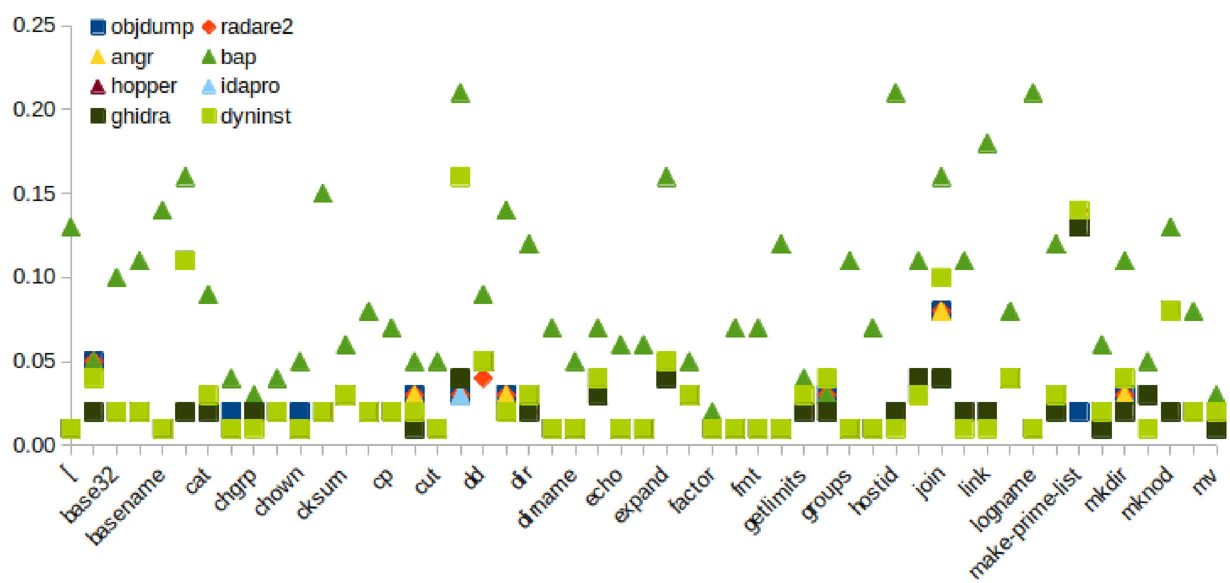

Figure 8.2: Ratio of grey instructions to white for different disassemblers.

The amount of white instructions per disassembler is an indication of how many instructions have been reached. objdump, radare2, angr, and Ghidra have similar numbers of white instructions. Meanwhile, BAP has smaller results in all these test cases since it does not employ any heuristics to solve the indirect branch problems caused by jump tables. The results for Dyninst are unstable because there are some instruction-recovery errors in the disassembly results.

8.2.3 Soundness Results

Most disassemblers are sound for most of the test cases. We find that Ghidra sometimes incorrectly recovers instructions. There are three other major exceptions.

First, BAP does not resolve indirect branches. Since BAP essentially reports an empty set of next addresses for indirect jump tables – whereas DSV wants to continue exploration – DSV reports a soundness issue. We marked these as missing instructions: the issue is not that BAP incorrectly recovers instructions, but that it misses instructions by "under-approximating" control flow.

Additionally, radare2 sometimes translates instructions to data. For example, in `dir` test case, radare2 disassembles the bytes `ff2552c72100` at address `3888` to data `.qword 0x90660021c75225ff`, which should be translated to a `call` instruction to the malloc function. This kind of mistranslation leads to missing instructions.

In some situations, Hopper is not capable to correctly determining the instruction boundaries. For example, in `dir` test case, at address `0xf2a8`, the disassembler should generate an instruction `sub r12d,0x1`. However, Hopper classifies it as data and continues the disassembly process from address `0xf2a9`.

Another exception is Dyninst. There are various examples showing that Dyninst involves errors in instruction recovery. These errors may cascade since incorrectly recovering instructions may also lead to incorrectly assessing which instruction addresses are to be disassembled. For instance, Dyninst cannot recover control flow for the `seq` test case from the Coreutils library since incorrectly recovered instructions lead to unrealistic paths.

Chapter 9

WinCheck: Formulation of Properties

We propose a bounded concolic model checker named WinCheck to analyze closed-source Windows executables. Our tool is capable of detecting three different kinds of memory-related errors, including buffer overflow, null-pointer dereference, and use after free.

In this chapter, we formally describe the three properties that can be verified using WinCheck and propose a model of the memory system. To start with, we introduce some types and functions which are useful in the formal definitions. Type $\mathbb{N}$ stands for natural numbers, and $\mathbb{B}$ refers to Bools. Since the concolic model checker supports 64-, 32-, and 16-bit address systems, the type of the concrete address is represented as $\mathbb{W}_n$, where $\mathbb{W}$ represents words (bit-vectors), and n indicates the number of bytes.

9.1 State Modeling

We have *symbolic* expressions of type $\mathbb{E}$ that have operators such as plus, minus, and times. The symbolic expressions have four types of operands: registers (notation $\texttt{RAX}, \texttt{RBX}, \ldots$), memory dereferences (notation $*[a, sz]$, where a is a symbolic expression representing the memory address and sz denotes the size of the memory region), immediate values, and bottom (notation $\perp$). The $\perp$ indicates that the corresponding value is undefined.

A state σ of type $\mathbb{S}$ stores symbolic expressions for registers, memory regions, and flags. The memory part of state σ, notation σ_{mem}, is modeled as a partial function, which employs a concrete memory address and a size, and gets the corresponding value from the memory model, either symbolic or concrete. If the corresponding value does not exist at the given address, then function σ_{mem} would return $\perp$. The partial function σ_{mem} is type-annotated as:

$$\sigma_{mem} :: \mathbb{W}_n \times \mathbb{N} \rightharpoonup \mathbb{E}$$

A symbolic expression is *resolvable* in a state σ if all its operands are assigned a concrete value in that state. Note that a symbolic expression containing $\perp$ typically is not resolvable. We use a partial function $\mathsf{resolve}$ to evaluate the concrete result from a symbolic expression in a state. If the symbolic expression is not *resolvable* in the state, the result of function $\mathsf{resolve}$ would be $\perp$.

$$\mathsf{resolve} :: \mathbb{E} \times \mathbb{S} \rightharpoonup \mathbb{W}_n$$

Functions mem_read and mem_write model respectively reading from and writing to memory. Function mem_read, of type $\mathbb{E} \times \mathbb{N} \times \mathbb{S} \mapsto \mathbb{E}$, takes as input a *symbolic* address, the size of the region to be read and the current state. As output, it produces a symbolic value. If the address is resolvable, the function will look up the resolved address in the current state. Otherwise, mem_read will return $\perp$.

Definition 9.1. Let a be an address, sz be the size in bytes, and let σ be a state. A *memory read*, notation $mem\_read(a, sz, \sigma)$, is defined as:

$$\mathsf{mem\_read}(a, sz, \sigma) = \begin{cases} \sigma_{mem}(resolve(a, \sigma), sz) & \text{if a is resolvable in } \sigma \\ \perp & \text{otherwise} \end{cases}$$

Function mem_write, of type mem_write :: $\mathbb{W}_n \times \mathbb{E} \times \mathbb{N} \times \mathbb{S} \mapsto \mathbb{S}$ takes as input a *concrete* address, a symbolic value to be written to memory, the size and the current state. In contrast to reading, memory writing always needs to happen to a concrete address. The state-space exploration algorithm (see Section 10.5) will have to ensure that any time a memory write happens, the pointer is resolvable. Its definition simply updates the partial function σ_{mem}.

To collect the memory writing information in a specific state, we define a function write_info, which fetches the next instruction to be executed in the current state, and provides all the memory regions that are written to by that instruction. For each region, it aims to resolve its address. If the address is resolvable, it adds a tuple (a, sz) to the returned set, indicating a resolved memory region with starting address a and size sz. If the address of a region is not resolvable, function write_info adds a bottom value $\perp$ to the set. Function write_info is type-annotated as:

$$\mathsf{write\_info} :: \mathbb{S} \mapsto \{(\mathbb{W}_n, \mathbb{N})?\}$$

Here, notation $(\mathbb{W}_n, \mathbb{N})?$ indicates an option type, i.e., either a value of type $(\mathbb{W}_n, \mathbb{N})$, or bottom. Similarly, function read_info is defined for memory reads and is of type $\mathbb{S} \mapsto \{(\mathbb{W}_n, \mathbb{N})?\}$.

Example 9.2. Let the current state be σ, $\sigma(rax) =$ `0x1006`. Let the next instruction be `mov qword ptr [rax + 8], 42`. Then write_info$(\sigma) = \{(\texttt{0x100E}, 8)\}$.

A statepart sp of type $\mathbb{SP}$ is either a register, a memory region, or a flag. We define a function eval to fetch the value stored inside the corresponding statepart in a state.

$$\mathsf{eval} :: \mathbb{S} \times \mathbb{SP} \mapsto \mathbb{E}$$

9.2 Memory-Related Properties

The concolic model checker is able to detect memory-related properties, including buffer overflow, use-after-free, and null-pointer dereference. The buffer overflow errors happen in the memory writing process; use-after-free errors occur during both reading and writing; null-pointer dereferences while reading.

Function buffer_overflow defines when a write executed by function mem_write is considered a buffer overflow.

Definition 9.3. A memory writing operation leads to buffer overflow, if and only if, at a state σ, for the memory writing operation with writing address a and size sz, there already exists a memory block in the state σ with a partial overlap.

$$\begin{aligned} &\text{buffer\_overflow}(\sigma) = \exists (a, sz) \in W \cdot \exists a', sz' \cdot \begin{cases} \sigma_{mem}(a', sz') \neq \bot \\ (a < a' \wedge a + sz > a') \vee (a' < a \wedge a' + sz' > a) \end{cases} \\ &\quad \textbf{where} \ \ W = \text{write\_info}(\sigma) \end{aligned}$$

In words, a buffer overflow is detected if a write occurs to a region (a, sz) that *partially overlaps* with a region (a', sz') in memory. Note that regions are added to memory either during a first write (e.g., a local variable in the stack frame is initialized) or when explicitly allocated on the heap (e.g., via a `malloc`).

Example 9.4. Let state σ be such that $\sigma(rax) = \texttt{0x1006}$, $\sigma_{mem}(\texttt{0x1010}, 8) = 0x3423$. Let the next instruction be `mov qword ptr [rax + 8], 42`. Then there exists a buffer overflow, since the new writing address `0x100E` with size 8 overlaps with the existing memory region $*[\texttt{0x1010}, 8]$ in σ_{mem}.

We define a function use_after_free to present the use-after-free error. If a memory record in the σ_{mem} model does not exist or is released following the execution of certain functions, such as `free`, then the memory reading/writing operation at the specific memory address would lead to a use-after-free error.

Definition 9.5. A memory operation leads to a user-after-free, if and only if, σ_{mem} model does not contain the record of the corresponding memory region.

$$\begin{aligned} &\text{use\_after\_free}(\sigma) = \exists (a, sz) \in A \cdot \neg\exists a', sz' \cdot \begin{cases} \sigma_{mem}(a', sz') \neq \bot \\ a' \leq a \\ a + sz \leq a' + sz' \end{cases} \\ &\quad \textbf{where} \ \ A = \text{read\_info}(\sigma) \cup \text{write\_info}(\sigma) \end{aligned}$$

Null-pointer dereference error happens when the memory reading/writing address is invalid, such as `NULL`, in a memory operation.

Definition 9.6. In a memory reading/writing operation, null-pointer dereference error occurs, if and only if, there exists a memory address a in the memory operation where a is `NULL`. We use a null_pointer_deref function to illustrate the null-pointer dereference.

$$\begin{aligned} &\text{null\_pointer\_deref}(\sigma) = \exists (a, sz) \in A \cdot \ a == \texttt{NULL} \\ &\quad \textbf{where} \ \ A = \text{read\_info}(\sigma) \cup \text{write\_info}(\sigma) \end{aligned}$$

Chapter 10

WinCheck: Algorithm

In the implementation of the WinCheck for binary files, we have encountered multiple challenges. These challenges are divided into three different categories: symbolic memory addresses, indirect branches, and state explosion problems. To handle each of these challenges, we develop various techniques, including a tracing-back system, user-adaptive concretization, and bounded loop handling.

In this chapter, we introduce how to resolve the symbolic memory address and indirect jump problems using a tracing-back system in Section 10.1. To ensure that a writing memory address is concrete, we adopt two different kinds of techniques to concretize a symbolic source. The first kind of technique, named constraint solving, is introduced in Section 10.2. Section 10.3 illustrates the second technique which is the user-guided concretization. We briefly describe how to prevent the state explosion problem in Section 10.4. Finally, the detailed execution step of the concolic execution is illustrated in Section 10.5.

10.1 Tracing-Back System

The base algorithm of WinCheck is a standard symbolic execution engine, performing a forwards state-space exploration. However, at some point during exploration, a state may be encountered that is too symbolic, in which case a trace-back may be triggered. We identify two causes for triggering a trace-back:

- A memory write occurs, but the region to which is written is unresolvable;
- The next value of the instruction pointer is unresolvable.

The first kind of tracing-back system is used to solve the symbolic memory-writing address problem. Listing 10.1 shows an example that triggers the tracing-back strategy by writing to an unresolvable memory address stored at the statepart `rsi` at state σ_2. After repeated tracing back, the concolic model checker reaches state σ_0 and halts since the unresolvable memory address is caused by `call` to an external function `strrchr`. Then how to handle the external function call at state σ_0 to ensure the execution could be carried out successfully is introduced in Sections 10.2 and 10.3.

Listing 10.1: A list of states that trigger the first kind of trace-back.

```
σ0 0x4000: call 0x41622c # 41622c <strrchr@GLIBC_2.2.5>
σ1 0x4005: mov rsi, rax
σ2 0x4007: mov qword ptr [rsi], 42
```

We apply another type of tracing-back system to handle the indirect branches caused by jump tables. In binary files, indirect jump addresses that are stored in jump tables can be accessed with the basic jump table address and corresponding jump table indices. In the concolic execution, the jump table index could be symbolic, which leads to the indirect jump address issue. For example, as illustrated in Listing 10.2, if the value of *rdi* is symbolic at state σ_0, then we cannot resolve the value of *rax* at the *jmp rax* instruction at state σ_4. Thus, WinCheck traces back until state σ_0 is reached. Then the indirect jump issue is solved using the algorithm introduced in algorithm [23].

Listing 10.2: A typical pattern for jump table without concrete index.

```
σ0 0x4705: cmp rdi, 4
σ1 0x4709: ja 799
σ2 0x470b: lea rax, [rip+0x20090e]
σ3 0x4712: mov rax, QWORD PTR [rdi+rax*1]
σ4 0x4716: jmp rax
```

During forwards state-space exploration, let $\tau = [\sigma_0, \ldots, \sigma_i, \ldots, \sigma_n]$ be the current trace, where each subsequent state executes a single instruction. At all times, there is exactly one trace from the initial state σ_0 to the current state σ_n. We use $\mathbb{T} = [\mathbb{S}]$ to denote the type of traces (lists of states).

Let σ_i be a state in which the next instruction assigns a symbolic value to a certain statepart *sp*, ensuring that $\mathsf{eval}(\sigma_{i+1}, sp) = \perp$. An *intervention* for tuple (σ, sp) is an action undertaken by the algorithm to prevent this specific symbolization. An intervention may be automated, e.g., the algorithm can choose to concretize values non-deterministically if it has established an upper bound to the statepart. An intervention may also consist of asking the user for input (e.g., when more information for an external function is required). Applying an intervention thus leads to a non-empty set of new states that is a subset of all concrete states represented by the more symbolic state σ_{i+1}. This effect cascades through the current trace, and thus applying an intervention produces a set of new traces of the form:

$$\tau' = [\sigma_0, \ldots, \sigma_i, \sigma'_{i+1}, \ldots \sigma'_n]$$

Example 10.1. An intervention on σ_0 in Listing 10.1 may set the value of `rax` to a concrete memory address after the execution. This will produce a new state σ'_1 in which the value of `rax` is concrete. Furthermore, the value of `rsi` is also resolvable after the the execution of instruction `0x4005: mov rsi, rax`. Then we get a new state σ'_2. Finally, in the execution of `0x4007: mov qword ptr [rsi], 42`, the value fetched from `rsi` is a resolvable memory address and the intervention in σ_0 in Listing 10.1 leads to a new trace $[\sigma_0, \sigma_1', \sigma_2']$.

Applying one intervention thus leads to a set of new traces, leading to new states $\{\sigma_n', \sigma_n'', \ldots\}$, each of which is "more concrete" than the original current state σ_n. The exact same holds when applying a list of *multiple* interventions subsequently. During a state-space exploration, one can choose to replace the current state σ_n with the set of new states – updating the current trace accordingly – and continue exploration from each of these. This action is undertaken by function intervene of type:

$$\mathsf{intervene} :: [\mathbb{S} \times \mathbb{SP}] \times \mathbb{T} \mapsto \{\mathbb{T}\}$$

Function intervene takes as input a list of interventions I and the current trace τ, and produces the set of new traces by applying each intervention subsequently.

Definition 10.2. Let $\tau = [\sigma_0, \ldots, \sigma_i, \ldots, \sigma_n]$ be the current trace, and let statepart *sp* evaluate to a symbolic value in the current state σ_n. A *trace-back* is a list of interventions I that successfully concretizes statepart *sp*:

$$\mathsf{trace\_back}(I, \tau, sp) \stackrel{\text{def}}{=} \forall \tau' \in \mathsf{intervene}(I, \tau) \ \cdot \ \mathsf{eval}(\sigma_n', sp) \neq \bot$$

Here, σ_n' is the last state of trace τ'. In words, a trace-back is a list of interventions that, when applied, ensures that the given statepart is no longer symbolic but has a concrete value for *all* new current states. Given the current trace τ, there may be different trace-backs, i.e., there may be different interventions possible. The algorithm aims to find a minimal trace-back, i.e., it aims to find the smallest possible intervention that concretizes a given statepart.

Example 10.3. Consider again Listing 10.1. When the concolic model checker encounters the instruction `mov qword ptr [rsi], 42` at address `0x4007`, it halts due to the unresolvable memory address stored in `rsi`. Suppose the current state is σ_2, the corresponding statepart is `rsi`, and the current trace is $\tau = [\sigma_0, \sigma_1, \sigma_2]$. Then after the execution of $\mathsf{trace\_back}([(\sigma_2, rsi)], \tau)$, we could get a new trace set $[\tau'] = \mathsf{intervene}([(\sigma_2, rsi)], \tau)$. For each of the trace τ' inside the trace set, suppose $\tau' = [\sigma_0', \sigma_1', \sigma_2']$, then the value of rsi at the state σ_2' is a resolvable memory address.

10.2 Intervention: Constraint Solving

The general concretization strategy is designed to handle the sources of the symbolic memory-writing addresses according to different types of these sources. As discussed in Section 10.1, the concolic model checker locates the sources for the symbolic memory addresses in memory writing operations using the tracing-back system. Then the concolic model checker concretizes these sources to ensure that the memory address is concrete in the memory-writing operation.

We employ the path constraint to a state to implement a general concretization procedure. For a specific state σ, there exists a unique path constraint π of type Π, which upholds the branching conditions of the trace to state σ. We use a function $\mathsf{path\_constraint}$ to extract the corresponding π from the trace to the state σ, function $\mathsf{path\_constraint}$ is of type $\mathbb{T} \mapsto \Pi$.

The major concretization process is carried out by a function named solve. The function solve uses the path constraint π and a symbolic expression as inputs and generates a concrete value set. The solve function is carried out by Z3 SMT Solver [32].

$$\mathsf{solve} :: \Pi \times \mathbb{E} \mapsto \{\mathbb{W}_n\}$$

The concretization technique is adaptive to different kind of symbolic sources. We use a function $\mathsf{concretize\_solve}$ to represent the concretize procedure. The function $\mathsf{concretize\_solve}$ is type-annotated as $\mathbb{S} \times \mathbb{SP} \mapsto \{\mathbb{S}\}$.

Definition 10.4. Let σ be the current state and τ be the current trace to σ. At state σ, statepart sp is evaluated to a symbolic value. Function $\mathsf{concretize\_solve}$ carries out the concretization process in a way:

$$\begin{array}{l}
\mathsf{concretize_solve}(\sigma, sp) \stackrel{\text{def}}{=} \{\sigma' \ \mid\ \mathsf{eval}(\sigma', sp) \in \mathsf{solve}(\pi, e) \wedge \forall_{sp' \neq sp} \cdot \mathsf{eval}(\sigma', sp') = \mathsf{eval}(\sigma, sp')\} \\
\textbf{where} \\
\quad \begin{array}{lcl} e & = & \mathsf{eval}(\sigma, sp) \\ \pi & = & \mathsf{path_constraint}(\tau) \end{array}
\end{array}$$

In words, an intervention based on constraint solving considers the set of all states σ' where statepart sp has been concretized according to the results of the solver, but the rest of the state remains untouched. Concretizing a state is one of the possible interventions executed by the algorithm.

10.3 Intervention: User-Guided Concretization

The concretization algorithm introduced in Section 10.2 is capable of concretizing the symbolic values in a symbolic expression based on currently available constraints. It may be the case that tracing back leads to an instruction that overwrites some statepart with an unconstrained unknown symbolic bottom value. This section discusses the second type of intervention for dealing with these cases. The sources of the symbolic values are *external functions* or *initial symbolic values* assigned at the starting point.

We have modeled certain external functions, such as `malloc`, `calloc`, and `free`. In the concolic model checker, the execution of these external functions would generate proper values for each register. Providing such a model for all external functions is infeasible since the number of external libraries is numerous.

The Windows calling convention dictates which registers are to be preserved by an external function call (callee-saved registers) and which registers are possibly overwritten (caller-saved registers). After execution of each external function call, all the caller-saved registers are set to symbolic values. Moreover, the contents of memory regions pointed to by any pointer passed as a parameter are overwritten with a symbolic value as well. As such, a trace-back may lead to a function call.

For example, consider function `strcpy`. Without intervention, the return value in register `rax` will simply be symbolic ($\perp$). If this symbolic pointer is – anywhere in future exploration – used to write to, then the algorithm will trace back to this function call. The user will be asked for information concerning function `strcpy`.

On the other hand, if the source of the symbolic value directly comes from the starting point, it is possible that the symbolic value still has a special meaning. For example, in the case of a standard C `main` function, the initial value in register `rdi` is the value of `argc`, which is the number of the arguments. Thus, if we need to concretize the symbolic value stored in `rdi`, we allow the user to set an upperbound to ensure that the concretized `argc` value is within a reasonable range.

The second type of intervention allows the user to concretize bottom values after external function calls and stateparts of the initial state. In this strategy, the user can define specific constraints for these cases. The user-defined constraint will be added to the path constraint, and the solve function will generate concrete values that satisfy the constraint if the corresponding symbolic value needs to be concretized.

For example, as shown in Listing 10.3, we define that after the execution of function __libc_start_main (the function that starts the C main function) the value of register `rdi` should be within the range $[0, 15]$, which indicates that the number of function arguments would be concretized in this range. As another example, the return value of `rax` is constrained to $(0, 14]$ after the execution of function getopt.

Listing 10.3: An example that shows the constraint for certain external functions.

```
__libc_start_main 0 <= RDI <= 15
getopt            0 < RAX <= 14
```

Some of the external functions would allocate a fresh memory address to specific stateparts. In this case, WinCheck provides another rule that allocates a fresh heap pointer for the specific stateparts after the execution of the external function designated by the user, as shown in Listing 10.4. The fresh heap pointer is concrete, and without further information, the size of the allocated memory region would be a default maximum memory region size.

Listing 10.4: Generate a fresh heap pointer after specific external function.

```
__errno_location RAX=fresh heap pointer
```

The value of caller-saved registers is made symbolic by default. However, some functions may preserve certain of these stateparts. A trace-back would then require the user to specify that a certain function does *not* overwrite a state part. Thus, users could define a rule for a specific external function that certain stateparts are unchanged after the execution of the function, as illustrated in Listing 10.5.

Listing 10.5: Reserve the value of certain statepart after an external function.

```
strlen RSI=unchanged
```

Finally, we allow the user to set constraints on the initial state. Without intervention, the initial state is entirely symbolic. A trace-back may lead back all the way to the initial state. This typically happens for the initial value of `rdi`, which indicates the number of the command-line arguments. For example, as shown in Listing 10.6, the value of `rdi` is set to between $[0, 6)$ in the concretization procedure after the starting point is hit.

Listing 10.6: The constraint for external environment expressions.

```
starting_point  0 < RDI <= 6
```

This user-adaptive concretization, on one hand, requires user interaction. However, the trace-back nature of the algorithm causes this interaction to be limited to exactly the information that is required to keep pointers concrete. On the other hand, it does not require a virtually unbounded number of external functions to be modeled.

10.4 Bounded Loop Handling

A substantial but common challenge in the concolic execution is the path explosion problem. To handle the problem, we prune some infeasible paths using the path constraint. Besides, in our implementation, the concolic model checker reduces memory usage by sharing unchanged states between multiple blocks. However, there still exists the situation that the construction fails due to running out of resources. A major reason is the loop.

For a bounded loop with concrete loop variables, we simply unroll the loop execution; however, a large loop count may lead to state space explosion. Moreover, an unbounded loop or a loop with symbolic loop variables may not be terminated. To solve this problem, we detect all the loops in the concolic execution and take a record of the counts of each loop's visit. If a count of a loop visited is larger than a pre-defined upperbound, then the loop will terminate directly. This algorithm is straightforward, and it straightforwardly ensures termination of concolic exploration.

10.5 Concolic Execution

The concolic model checker takes binary files as inputs and checks the memory-related properties, as defined in Chapter 9, on the binaries. This concolic model checking process can be divided into concolic execution and property checking processes (see Algorithm 2).

Starting from the initial trace τ_0 that only contains state σ_0, the concolic model checker carries out forward exploration. The algorithm considers the current state σ_n. The next instruction to be executed is determined by the current state (by its instruction pointer).

If the next instruction at state σ_n is not a memory-writing instruction or the memory-writing address is concrete, then the algorithm 1.) performs property checking (Line 12) and then 2.) proceeds with forwards exploration (Lines 13 to 16). Line 12 considers the current state σ_n and verifies properties such as the ones detailed in Chapter 9. As soon as one of the properties does not hold, exploration for the current trace τ is halted, and trace τ is presented as a counterexample. The function exec_step represents a single non-deterministic step of the corresponding instruction at state σ_n:

$$\mathsf{exec\_step} : \mathbb{S} \mapsto \{\mathbb{S}\}$$

For each of the new state σ_{n+1}, we append σ_{n+1} to the trace τ and construct a new trace τ'. Then the state exploration continues on the trace τ'.

If the current instruction does write to a symbolic memory address, an intervention must occur (Lines 5 to 11). For all memory regions (a, sz) in *write_set*, if the corresponding address a is not resolvable in σ_n, we add (a, sz) to set *symbs*. If *symbs* is not empty, WinCheck carries out the trace-back process for all the unresolvable memory addresses and obtains a list I of all interventions necessary to perform a successful trace-back (see Definition 10.2). This produces a new set of traces τ_set. For all the new trace τ' insides τ_set, we continue on the state exploration process using the concolic_exploration function.

Algorithm 2 Concolic execution that explores the whole state space.

function CONCOLIC_EXPLORATION(τ)
    $\sigma_n \leftarrow$ last state of τ
    $write\_set \leftarrow \mathsf{write\_info}(\sigma_n)$
    $symbs \leftarrow \{(a, sz) \in write\_set \cdot \mathsf{resolve}(a, \sigma_n) = \bot\}$
    if $symbs \neq \emptyset$ **then**
        Obtain I such that $\forall sp \in symbs \cdot \mathsf{trace\_back}(I, \tau, sp)$
        $\tau\_set \leftarrow \mathsf{intervene}(I, \tau)$
        for all $\tau' \in \tau\_set$ **do**
            $\mathsf{concolic\_exploration}(\tau')$
        end for
    else
        $\mathsf{property\_checking}(\sigma_n)$
        for all $\sigma_{n+1} \in \mathsf{exec\_step}(\sigma_n)$ **do**
            $\tau' \leftarrow \tau @ [\sigma_{n+1}]$
            $\mathsf{concolic\_exploration}(\tau')$
        end for
    end if
end function

Chapter 11

WinCheck: Experimental Results

In this chapter, we first apply WinCheck on some closed-source Windows executables and present the results (see Section 11.1) to demonstrate that it is capable of analyzing binaries without source code availability. Then we apply WinCheck on all the 97 test cases in the Coreutils-5.3.0 library for Windows to evaluate performance.

We execute all the test cases illustrated in this section on a host with an Intel Core i7-7500U CPU and 16GB RAM.

11.1 Closed-Source Windows Executables

The binaries used in this section are taken from a standard Windows 10 distribution. This means they were heavily optimized, and we have no availability over any details on compiler settings, the build process, etc. The calling conventions of these binaries are unknown, and they are – often even intra-binary – mixed between, e.g., callee vs. caller stack clean-up.

Table 11.1 shows the results. The first column of the table is the name of the binary, the second column shows the number of instructions reached by WinCheck, the third column indicates the total number of paths explored. The fourth column provides the total number of negative paths. We consider a path to be a negative if it ends in a state with either a buffer overflow, a use-after-free, or a null-pointer dereference. A negative requires further manual inspection. In Section 11.5 we provide a discussion on true vs. false negatives. The fifth column provides the number of paths that contained a state in which an instruction read from uninitialized memory, but none of the other pointer-related issues occurred that would classify it as a negative. Column 6 provides the number of instructions that performed an unresolvable indirect branch, i.e., either a call or a jump whose jump target could not be concretized.

Finally, Column 7 provides an approximation of the number of instructions not reached. The actual number of reachable instructions is undecidable. When running a disassembler such as IDA Pro, this provides an estimate of all instructions in the binary. However, these instructions are not necessarily reachable, i.e., that number over-approximates the actual number of reachable instructions. However, to give an impression of the part of the binary that is actually reached by WinCheck, we report in Column 7 the total number of instructions

Table 11.1: Memory security verification results for closed-source Windows executables.

Exec name	# of reached instrs	# of paths	# of negatives	# of uninitialized	# of unresolved indirects	# of unreached instrs
ARP.EXE	2825	996	8	9	3	777
HOSTNAME.EXE	1037	241	0	10	3	110
clip.exe	3642	1078	0	13	10	1732
ftp.exe	3898	1423	0	9	3	6405
logman.exe	6351	2321	0	101	34	11507
msconfig.exe	570	174	0	20	3	16615
ndadmin.exe	1625	591	0	30	5	209
netsh.exe	5028	1696	0	206	5	6125
ping6.exe	3002	1443	0	194	2	4437
replace.exe	1438	245	0	14	18	1034

reported by IDA Pro minus the number of instructions reached by WinCheck.

For the closed-source Windows executables, the number of reached instructions is roughly 38%. In Section 11.3 we provide a more detailed discussion on the reasons for unreached instructions.

For these Windows executables, `ARP.EXE` is the only one for which we encountered negatives. Detailed tracing back information for these errors is provided in a .log file. Listing 11.1 provides an example of a candidate for a null-pointer dereference. The algorithm traces back the null-pointer, traverses various instructions across function boundaries, and finds the sources of the null-pointer. We could establish that the negative was a false negative due to the infeasibility of the path.

Listing 11.1: Null-pointer dereference in the execution of ARP.EXE.

```
Error: 0x401596 mov eax,dword ptr [ecx]
  Null pointer dereference at address 0x0

Trace back to ['rcx'] after 0x401596: mov eax,dword ptr [ecx]
Trace back to ['4294966488'] after 0x401592: mov ecx,dword ptr [esp+0x1c]
Trace back to ['rdi'] after 0x40152b: push edi
Trace back to ['rdi'] after 0x40150a: mov edi,edi
Trace back to ['rbx'] after 0x40376d: mov edi,ebx
Trace back to [] after 0x40375f: xor ebx,ebx
```

11.2 Coreutils Library

To illustrate the applicability and scalability of the concolic model checker, we apply it to the Coreutils-5.3.0 library. Similar to Table 11.1, for each test case in the Coreutils library, we respectively collect the information regarding the number of reached instructions, number of paths traversed, number of negative paths, number of unresolved indirect jumps, and number of unreached instructions.

As shown in Table 11.2, three of the test cases, including `cp.exe`, `mv.exe`, and `shred.exe`, expose negative paths during the execution. The negative paths in `cp.exe` and `mv.exe` are caused by null-pointer dereference errors. In `shred.exe` a possible buffer overflow was encountered.

Listing 11.2 provides more information on the negative encountered in `shred.exe`. A buffer-overflow error is reported at address `0x403009` with instruction `mov dword ptr [edx],ebx`. The reason lies in that the memory address that is stored in `edx` is `0x400` at that address. We repeatedly trace back and finally find that the value of `edx` comes from the instruction `mov ebx,0x400` at address `0x402961`. Writing to the memory address `0x400` at address `0x403009` would lead to a buffer overflow since a partially overlapping memory region already exists in the memory model.

Listing 11.2: Buffer overflow error.

```
Error: 0x403009 mov dword ptr [edx],ebx
   Buffer overflow at address 0x400

Trace back to ['rdx'] after 0x403009: mov dword ptr [edx],ebx
Trace back to ['4294965620'] after 0x402f46: mov edx,dword ptr [esp+0x24]
Trace back to ['rdi'] after 0x4029a4: mov dword ptr [esp+0x0],edi
Trace back to ['4294965612'] after 0x40b2ba: mov edi,dword ptr [esp+0x24]
Trace back to ['rbx'] after 0x40b1e7: mov dword ptr [esp+0x1c],ebx
Trace back to ['rbx'] after 0x402961: mov ebx,0x400
```

11.3 Unreached Instructions

As shown in Table 11.2, there are many unreached instructions for each of the test cases. We make a deep analysis of all the unreached instructions and divide the situation into three different categories.

First, while most of the functions in a binary file are visited using direct or indirect jump instructions, some of the functions are called implicitly. For example, as shown in Listing 11.3, the address `0x401c20` is pushed into the stack after the execution of `push 0x401c20`

Table 11.2: WinCheck results for the Coreutils library.

Exec name	# of reached	# of paths	# of negatives	# of uninitialized	# of unresolved indirects	# of unreached
[.exe	2054	663	0	2	4	9590
basename.exe	1973	875	0	34	4	1475
cat.exe	3023	1292	0	42	4	7444
chgrp.exe	6897	1954	0	14	8	8755
chmod.exe	6566	2029	0	172	11	9385
chown.exe	8239	2658	0	210	4	7798
chroot.exe	1693	722	0	10	4	1913
cksum.exe	1637	637	0	5	4	1992
comm.exe	2614	904	0	14	4	1092
cp.exe	5561	1473	15	10	6	19461
csplit.exe	4642	1746	0	20	6	10512
cut.exe	1739	571	0	24	6	5222
date.exe	7723	2497	0	28	2	8402
dd.exe	5391	2025	0	4	4	10170
df.exe	8448	2896	0	6	4	6458
dir.exe	6730	1725	0	2	7	21050
dircolors.exe	2872	1213	0	32	6	1633
dirname.exe	1981	873	0	27	4	1568
du.exe	9204	3023	0	8	4	12262
echo.exe	1257	513	0	2	4	2204
env.exe	1574	569	0	4	4	1798
expand.exe	2224	721	0	16	4	1861
expr.exe	3666	1223	0	2	4	8598
factor.exe	2395	1013	0	2	4	2720
false.exe	316	16	0	2	4	1723
fmt.exe	2901	848	0	2	4	2116
fold.exe	2308	955	0	5	4	1607
gdate.exe	7723	2497	0	28	2	8402
gecho.exe	1257	513	0	2	4	2204
ginstall.exe	11552	3584	0	17	2	15193
gln.exe	5827	1702	0	9	8	13483
gmkdir.exe	4852	1470	0	17	4	7314
grmdir.exe	2145	643	0	2	4	1408
gsort.exe	7876	2633	0	2	4	10065
head.exe	2867	934	0	34	9	10025
hostid.exe	2176	885	0	25	4	1601
hostname.exe	1577	640	0	6	4	1699
id.exe	2488	655	0	18	5	1610
install.exe	11552	3584	0	17	2	15193
join.exe	4559	1886	0	12	5	2216
kill.exe	2481	1031	0	5	0	1896
link.exe	2499	921	0	19	4	2562
ln.exe	5827	1702	0	9	8	13483
logname.exe	2025	764	0	38	4	1510
ls.exe	6739	1726	0	2	7	21041
md5sum.exe	1744	474	0	1	6	5825
mkdir.exe	4852	1470	0	17	4	7314
mkfifo.exe	2050	681	0	12	9	8549

mknod.exe	3243	1134	0	27	9	8544
mv.exe	8235	2215	9	5	14	19871
nice.exe	1655	712	0	3	4	2070
nl.exe	3257	918	0	9	12	9392
nohup.exe	2663	1080	0	13	4	1651
od.exe	4123	1735	0	49	4	10536
paste.exe	2329	1024	0	15	4	1718
pathchk.exe	2300	743	0	6	8	7700
pinky.exe	5763	1589	0	8	6	6336
pr.exe	2228	879	0	2	4	12248
printenv.exe	1194	583	0	2	4	1985
printf.exe	3132	1224	0	7	5	5092
ptx.exe	4956	2035	0	7	4	16299
pwd.exe	4600	1328	0	2	4	6043
readlink.exe	4397	1050	0	36	9	5914
rm.exe	7016	2192	0	20	4	14489
rmdir.exe	2145	643	0	2	4	1408
seq.exe	2142	816	0	23	11	1576
setuidgid.exe	2336	878	0	20	4	1459
sha1sum.exe	1735	468	0	1	5	5834
shred.exe	6341	1928	1	49	4	9827
sleep.exe	1884	691	0	3	4	2243
sort.exe	7876	2633	0	2	4	10065
split.exe	3597	1099	0	175	5	9105
stat.exe	5001	1152	0	14	5	8028
stty.exe	3314	1007	0	23	4	3799
su.exe	4734	1150	0	17	5	9281
sum.exe	1258	365	0	2	7	4855
sync.exe	1354	621	0	2	4	1761
tac.exe	3346	1315	0	11	4	14282
tail.exe	4829	1644	0	10	4	10987
tee.exe	2064	891	0	9	4	1305
test.exe	1168	374	0	26	5	9251
touch.exe	7173	2272	0	30	4	6323
tr.exe	4072	1641	0	50	6	2990
true.exe	316	16	0	2	4	1723
tsort.exe	2297	1060	0	16	4	1896
tty.exe	1931	564	0	31	4	1228
uname.exe	2666	755	0	11	5	2250
unexpand.exe	2811	1171	0	3	4	1475
uniq.exe	3048	1373	0	8	4	1988
unlink.exe	1681	754	0	39	4	1957
uptime.exe	2821	851	0	39	4	2420
users.exe	2801	860	0	41	4	3339
vdir.exe	5621	1348	0	2	4	22156
wc.exe	4519	1231	0	2	4	6171
who.exe	4850	975	0	13	4	7801
whoami.exe	2231	817	0	38	4	1480
yes.exe	1407	645	0	2	4	1762

instruction. Then in the execution of the function at address `0x40401c`, the function ___CxxUnhandledExceptionFilter is passed on as a function pointer and is implicitly called. However, since the definition of the function at address `0x40401c` is imported from a Windows DLL file named `api-ms-win-core-errorhandling-l1-1-1 .dll`, we cannot model the detail of the function. Thus we do not have an explicit call of the function ___CxxUnhandledExceptionFilter, and all the corresponding instructions in this function are skipped.

Listing 11.3: Implicitly called function.

```
0x401c70: push 0x401c20
0x401c75: call dword ptr [0x40401c]
...
0x401c20:     LONG __stdcall __CxxUnhandledExceptionFilter(struct
   _EXCEPTION_POINTERS *ExceptionInfo)
...
0x40401C ;    Imports from api-ms-win-core-errorhandling-l1-1-1.dll
0x40401C ;    LPTOP_LEVEL_EXCEPTION_FILTER (__stdcall
   *SetUnhandledExceptionFilter)(LPTOP_LEVEL_EXCEPTION_FILTER
   lpTopLevelExceptionFilter)
```

Second, some of the memory content is initialized at linking time, and we cannot get the corresponding value in a static analysis process. For instance, in HOSTNAME.EXE, we encounter an unresolved indirect jump at address `0x401b31`, where the value of `esi` is symbolic, as shown in Listing 11.4. After tracing back, we find the value of `esi` comes from `mov esi, dword ptr [0x403380]` instruction at address `0x401b23`. Further manual inspection on the section information of HOSTNAME.EXE showed that address `0x403380` is located at .data section, and the memory content at address `0x403380` has not been initialized yet. Thus we cannot concretize the value of `esi` or find a solution to the symbolic jump address using a static analysis method, and certain region at the binary file is not reachable in the concolic execution.

Listing 11.4: Dynamically linking memory content.

```
0x401b23: mov  esi, dword ptr [0x403380]
0x401b29: mov  ecx, esi
0x401b2b: call dword ptr [0x404108] ; _guard_check_icall_nop(x)
0x401b31: call esi
```

Finally, since our tool is implemented using bounded model checking, we will terminate a loop if the pre-defined bound is exceeded. The decision also leads to unreached instructions in certain cases. For example, in HOSTNAME.EXE, the instruction at address `0x40162D` is never visited since the condition has never been satisfied for the `jnz 0x4016C8` instruction at address 0x401627. Thus all the following instructions are not reachable during the bounded model checking.

Listing 11.5: Unreached instructions caused by conditional jumps.

```
0x401627: jnz    0x4016C8
0x40162D: lea    ecx, [ebp+var_4]
0x401630: call   sub_401550
0x401635: test   eax, eax
```

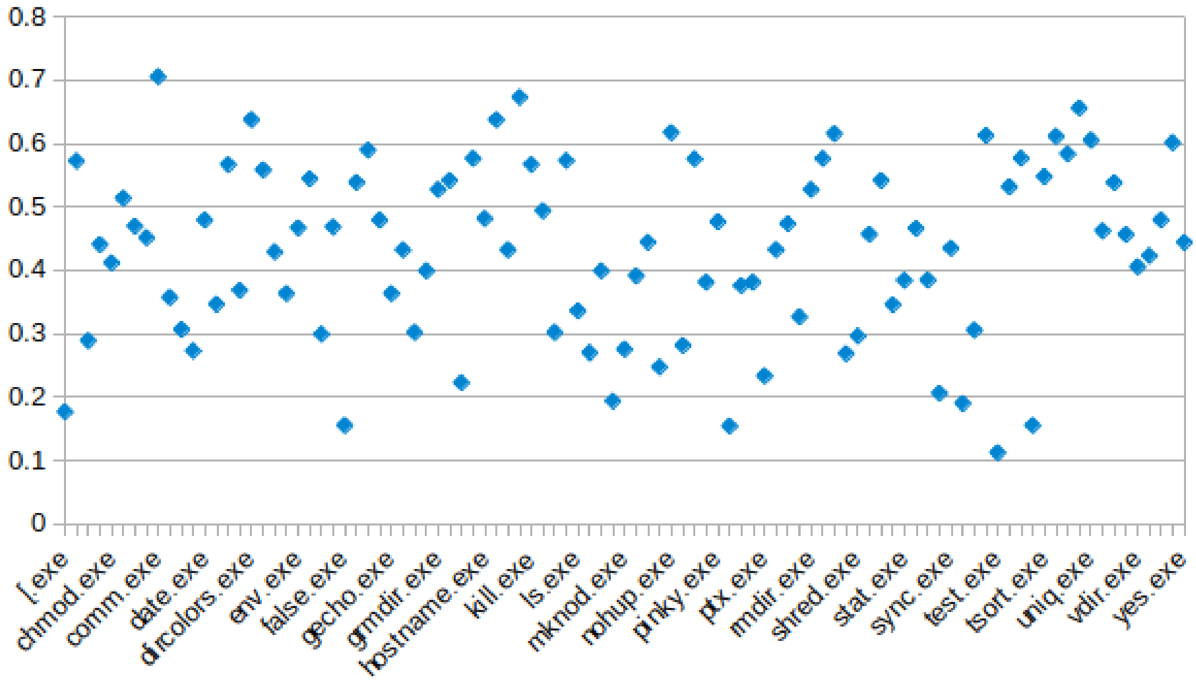

Figure 11.1: Ratio between # of reached instructions vs. # of total instructions.

As can be seen from Figure 11.1, the ratio between the number of reached instructions vs. the number of total instructions varies from 11% to 71%. Test case `test.exe` has the lowest ratio while `comm.exe` shares the highest ratio. On average, the ratio is around 42%, which means WinCheck could detect around 42% of the total instructions. After a deep inspection, we find that the sources of the unreached instructions are mainly implicit called functions and dynamically allocated memory content. For example, in `test.exe`, the number of total instructions is 10419. Inside that, 9096 instructions are directly or indirectly caused by implicit called functions, unresolved indirect jumps, and dynamically allocated content. To be more specific, 1031 instructions (divided into 87 instruction blocks) are not reached since there are no explicit entries to them, and the left 8065 instructions are not reached because the entries to these instructions are located in the 1031 directly unreached instructions. Then we can get the conclusion that the reachability ratio could be increased with a refined model of the external functions.

11.4 Miscellaneous

In the concolic model checking process, the disassembly process is carried out by IDA Pro 7.6, which has the best performance on the recursive-traversal disassembly on Windows executables. We have once considered using angr [102] to disassemble the binaries; however, there are some bugs in the disassembled results. For example, angr sometimes disassembles data to instructions. In the disassembly process of the HOSTNAME.EXE, angr disassembles the byte sequence at address 0x401000 as an instruction push eax, while IDA Pro translates the byte sequence to a data struct _EXCEPTION_POINTERS which is of type ExceptionInfo.

Besides, in the reconstruction of the control flow, angr also makes some errors in the static analysis. For instance, for basename.exe in Coreutils library, angr decodes the following instructions:

```
0x4042ca: and  byte ptr [0x61202c73], ah
0x4042ce: and  byte ptr [ecx + 0x6e], ah
0x4042d1: and  byte ptr fs:[edi + 0x74], ch
```

Meanwhile, the disassembled results from IDA Pro is as follows.

```
.text:004042ca and    ds:61202c73h, ah
.text:004042d0 outsb
.text:004042d1 and    fs:[edi+74h], ch
```

After repeated checking on the corresponding byte sequence, we are certain the IDA Pro generated the correct results.

11.5 Discussion

We here provide some discussion on the nature of applying state exploration tools to Windows executables.

Completeness In this context, completeness means that the state space exploration is over-approximative: at least all states are visited. It is clear from the results in Section 11.2 that this is not the case: WinCheck is not complete. This has three major causes, which have been discussed in more detail in Section 11.3. First, *bounded* model checking is under-approximative. Second, an unresolved indirect jump may lead to unreached instructions. A third cause is more tricky: it may be the case that a function pointer is stored in a statepart that is read by an external function (a parameter register or a global part of the state). That external function may then call the function pointed to. The exploration is complete *modulo* these issues, i.e., if the bound is not hit, if all indirect branches can be resolved, and if there are no callbacks executed by external functions, the full state space is explored. We argue that specifically, the last issue is inherent in the verification of closed-source binaries.

Soundness Conversely, soundness means that any state reached is actually reachable from the entry point. Of course, soundness depends on proper user input. Since WinCheck performs a standard forward exploration from the entry point of the executable, it is sound *modulo* the user input. We argue that in the exploration of closed-source binaries, soundness is a more desirable property than completeness: since a manual analysis of candidate negatives is hard, time-consuming, and expensive, any reported path should be relevant.

True vs. false negatives Since the exploration is sound, every negative path that is reported is an actual reachable path. However, still, a manual analysis is required to see if the reported negative is a true negative. As an example, we consider paths reporting an uninitialized read. These reachable paths truly do an uninitialized read, but whether that is an actual bug is a follow-up question. Manual inspection showed that many of these paths actually perform a technique introduced by a compiler called stack probing, where parts of the local stack are read-then-discarded to ensure these memory regions are paged properly. So indeed, the uninitialized read occurs, but this does not constitute a true negative.

User interaction The trace-back mechanism thwarts full automation. However, the information requested is minimized to necessary information only. While doing the case studies, we have maintained a file storing external function information (see Section 10.3) for various external functions shared between the executables. For example, functions regarding string operations or file- and memory management are included. This allowed us to perform all experiments with minimal user interaction.

Chapter 12

Conclusions and Future Work

In this dissertation, multiple efforts have been presented. The first two efforts aim to reduce the TCB of binary verification; meanwhile, the latter shows that binary verification can be carried out with a smaller TCB since it is based on a disassembly process that has been validated with the second effort.

The dissertation first presents a methodology called OPEV that provides a high assurance on the equivalence between OCaml and PVS specifications. OPEV employs an intermediate type system to capture the commonality of the subset of OCaml and PVS and to generate test cases for both OCaml and PVS implementations. The reliability of the validation is ensured by executing large-scale stress tests and automatically proving test lemmas using generic PVS strategies. OPEV generates more than three hundred thousand test cases and proofs. We demonstrate the OPEV methodology using two case studies, namely a manual OCaml-to-PVS translation and a Sail-to-PVS parser. OPEV significantly increases our trust in the translations.

The dissertation, then, introduces a definition for soundness of the output of a disassembler w.r.t. the original binary. We propose DSV, a tool for validating whether a binary has been correctly disassembled. Disassembly is a challenging and undecidable problem that lies at the base of various research in reverse engineering, formal verification, binary hardening, and security analysis. Even state-of-the-art disassemblers that have been elaborately designed and tested have soundness issues, such as whether a disassembly accurately reflects the semantical behavior of the binary under investigation. DSV finds incorrectly disassembled instructions and assesses whether the disassembler under investigation could determine at which addresses instructions need to be recovered correctly. DSV has been applied to validate the output of eight state-of-the-art disassembler tools on 102 binaries of the Coreutils library. Soundness issues were exposed, ranging from incorrect instruction recovery to incorrectly recovered control flow of the binary (leading to missing instructions).

DSV does not assume the existence of ground truth in the form of source code, an LLVM representation, or debugging information. We, therefore, necessarily make assumptions and aim to provide an explicit insight into the TCB. The TCB of DSV contains two key assumptions. First, we assume that the proposed way of loosely comparing byte sequences allows DSV to decide whether a single byte sequence correctly corresponds to a single instruction. Second, DSV employs concolic execution leaving certain parts, such as the stack pointer, concrete. It is assumed that leaving these parts concrete does not influence the reachability of instruction addresses.

The dissertation finally introduces a concolic model checker named WinCheck to detect memory-related errors in Windows executables. Detecting and eliminating memory-related errors in the development stage of binary files is an objective that is pursued by many in academia and industry alike. Various techniques, including testing, symbolic execution, and formal verification, have been developed to validate source code and binary executables. However, the state-of-the-art does not provide an off-the-shelf model checker that can directly be applied to Windows executables.

The main novelty of WinCheck is that it aims to leave any pointer-related information concrete, but the rest of the state as symbolic as possible. If need be, WinCheck will trace back to a point where an intervention can concretize the current state to ensure this characteristic. Concretization can occur automatically through constraint solving or by asking the user for specific information regarding external functions or the initial state. By keeping memory addresses concrete, the pointer-aliasing problem becomes decidable. Moreover, this allows resolving various indirect branches (dynamically computed jumps), typically a challenge in binary verification. Finally, concolic execution allows various memory-related properties to be easily verified, such as use-after-free, uninitialized reads, or buffer overflows.

To show the functionality and performance of WinCheck, we apply the model checker to two different kinds of test cases. We employ WinCheck on Windows closed-source binaries to show the functionality and compatibility of the concolic model checker. Moreover, we apply our tool to the Coreutils library – compiled on Windows – to analyze and show the performance of the concolic model checker.

12.1 Future Work

For each of the contributions mentioned above, there exist many future research orientations. We propose some future research directions in this section.

12.1.1 OPEV's Extension with Proof Automation Rules

Currently, OPEV handles a subset of OCaml types and pure functions. In the future, we aim to extend the functionality of OPEV and incorporate more test generation rules for it. We also intend to increase automation in the proof process of OPEV. These enhancements would allow us to translate multiple mainstream instruction sets (ISA) specifications written in Sail into PVS [106], a necessary step to reason about the binary code of these architectures in PVS. For instance, the methodology of lifting ARMv8 binaries into PVS7 [106] based on translating ARM specification language ASL [86] into PVS is interesting to us to generalize for other architectures. It allows the translation of the system binary code of ARMv8 into

PVS, based on PVS generic theories, theory parameters, and dependent types, in place of monad theory. Therefore, our work would open the door for more future research to verify the binary code of several mainstream instruction sets based on translating Sail ISAs specifications into the prototype verification system PVS.

12.1.2 DSV's Binary Exploration and Supporting More ISAs

DSV essentially is a binary exploration tool. We argue that DSV demonstrates that the combination of bounded model checking and concolic execution is very applicable in the context of stripped binaries as it mitigates the complexity of some fundamental issues. Even though its current version solely focuses on the validation of disassembly, we aim to use the core algorithm and concepts of DSV for other binary exploration efforts. For example, We aim to use DSV for validating the correctness of generated control flow and call graphs and generally for exposing "weird" edges [101] and security vulnerabilities in binaries. Currently, DSV is restricted to binaries with the x86-64 format. Since our formal definition is general, we intend to extend our implementation and validation efforts to other ISAs, such as ARM.

12.1.3 Full Exploration and New Functionalities in WinCheck

In WinCheck, a key limitation is the bounded nature, which limits the number of instructions reached. For aggressively optimized closed-source binaries such as those found on a Windows machine, we argue that an over-approximative approach such as deriving loop-invariants, or fully symbolic execution, does not scale to realistic executables or Windows DLLs. In the near future, we, therefore, aim to find a midway between bounded space state exploration and fully symbolic state-space exploration that allows full exploration of real-world Windows executables.

WinCheck verifies pointer-related properties in sequential code. Another interesting future direction is, therefore, to boost WinCheck to verify properties of concurrent code. For instance, WinCheck could be enhanced to verify whether the execution of a binary file in concurrent mode would cause a deadlock or not.

WinCheck currently focuses on low-level properties such as those pertaining to pointer usage. An interesting future direction is to extend WinCheck to verify higher-level properties such as information flow security [92] (on Windows binaries). WinCheck's underlying technique, i.e., user-guided concretization, would facilitate the incorporation of new functionalities. Besides, we could incorporate decompilation from Windows binaries to corresponding source code into WinCheck to verify higher-level properties; and the soundness of this decompilation process could also be validated using an extended DSV.

Bibliography

[1] Fog Agner. Calling conventions for different C++ compilers and operating systems. *http://www.agner.org/optimize/calling_conventions.pdf*, 2014.

[2] 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. doi: 10.1007/978-3-319-43144-4_3.

[3] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E Porter. X86-64 instruction usage among C/C++ applications. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 68–79, 2019. doi: 10.1145/3319647.3325833.

[4] Xiaoxin An, Amer Tahat, and Binoy Ravindran. A validation methodology for OCaml-to-PVS translation. In *NASA Formal Methods Symposium*, pages 207–221. Springer, 2020. doi: 10.1007/978-3-030-55754-6_12.

[5] Xiaoxin An, Freek Verbeek, and Binoy Ravindran. DSV: Disassembly soundness validation without assuming a ground truth. In *NASA Formal Methods Symposium*. Springer, 2022.

[6] Xiaoxin An, Freek Verbeek, and Binoy Ravindran. WinCheck: A concolic model checker for pointer-related properties on Windows binaries. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, December 2022. Under review.

[7] 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.

[8] Sharon Barner, Cindy Eisner, Ziv Glazberg, Daniel Kroening, and Ishai Rabinovitz. Explisat: Guiding sat-based software verification with explicit states. In *Haifa Verification Conference*, pages 138–154. Springer, 2006. doi: 10.1007/978-3-540-70889-6_11.

[9] 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.

[10] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on Uppaal. *Formal methods for the design of real-time systems*, pages 200–236, 2004. doi: 10.1007/978-3-540-30080-9_7.

[11] 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. doi: 10.1145/2024569.2024572.

[12] Dirk Beyer and M Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011. doi: 10.1007/978-3-642-22110-1_16.

[13] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers, Academic Press preprint*, 58, 2003.

[14] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *International Conference on Interactive Theorem Proving*, pages 131–146. Springer, 2010. doi: 10.1007/978-3-642-14052-5_11.

[15] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011. doi: 10.1007/978-3-642-22110-1_37.

[16] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, volume 8, pages 209–224, 2008.

[17] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, 2012. doi: 10.1109/SP.2012.31.

[18] FT Chan, Tsong Yueh Chen, IK Mak, and Yuen-Tak Yu. Proportional sampling strategy: Guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775–782, 1996. doi: 10.1016/0950-5849(96)01103-2.

[19] Xiaobo Chen, Dan Caselden, and Mike Scott. New zero-day exploit targeting internet explorer versions 9 through 11 identified in targeted attacks. *Fireeye blog*, 26, 2014.

[20] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011. doi: 10.1145/1961295.1950396.

[21] 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. doi: 10.1109/QEST.2004.1348056.

[22] Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995. doi: 10.1002/spe.4380250706.

[23] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40(2-3):171–188, 2001. doi: 10.1109/WPC.1999.777758.

[24] 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. doi: 10.1007/s100090050046.

[25] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *International conference on computer aided verification*, pages 359–364. Springer, 2002. doi: 10.1007/3-540-45657-0_29.

[26] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, 2011. doi: 10.1002/stvr.415.

[27] 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. ACM, 2000. ISBN 1-58113-202-6. doi: 10.1145/351240.351266.

[28] Mirko Conrad. Testing-based translation validation of generated code in the context of IEC 61508. *Formal Methods in System Design*, 35(3):389–401, 2009. doi: 10.1007/s10703-009-0082-0.

[29] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 351–366, 2007. doi: 10.1145/1323293.1294295.

[30] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Citeseer, 2001.

[31] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656. IEEE, 2016. doi: 10.1109/SANER.2016.43.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[33] Bjorn De Sutter, K De Bosschere, Peter Keyngnaert, and Bart Demoen. On the static analysis of indirect control transfers in binaries. In *Proceedings of the international conference on parallel and distributed processing techniques and applications*, volume 2, pages 1013–1019, 2000.

[34] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. doi: 10.1145/360933.360975.

[35] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE transactions on Software Engineering*, SE-10(4):438–444, 1984. doi: 10.1109/TSE.1984.5010257.

[36] 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. doi: 10.1007/978-3-540-93900-9_10.

[37] GCC, the GNU Compiler Collection. https://gcc.gnu.org/. Accessed: 2022-4-28.

[38] GNU Binutils. https://www.gnu.org/software/binutils. Accessed: 2022-4-28.

[39] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005. doi: 10.1145/1064978.1065036.

[40] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *The Network and Distributed System Security (NDSS) Symposium*, volume 8, pages 151–166, 2008.

[41] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012. doi: 10.1145/2090147.2094081.

[42] Michael JC Gordon et al. The HOL system description. *Cambridge Research Centre, SRI International*, 2010.

[43] Kathryn E Gray, Peter Sewell, Christopher Pulte, Shaked Flur, and Robert Norton-Wright. The Sail instruction-set semantics specification language. Technical report, Cambridge University, 2017.

[44] 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. doi: 10.1007/978-3-642-32347-8_8.

[45] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 78–88, 2012. doi: 10.1145/2338965.2336763.

[46] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 2:971–978, 1994.

[47] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001. doi: 10.1016/S0950-5849(01)00189-6.

[48] 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. doi: 10.1145/2893177.

[49] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997. doi: 10.1109/32.588521.

[50] Hopper: The macOS and Linux Disassembler. `https://www.hopperapp.com/`. Accessed: 2022-4-28.

[51] M Hou, AC Pugh, and GE Hayton. Generalized transfer functions and input-output equivalence. *International Journal of Control*, 68(5):1163–1178, 1997. doi: 10.1080/002071797223262.

[52] IDA Pro: State-of-the-art binary code analysis tools. `https://www.hex-rays.com/ida-pro/`. Accessed: 2022-4-28.

[53] Intel 64 and IA-32 Architectures Software Developers Manual. `https://cdrdv2.intel.com/v1/dl/getContent/671200`. Accessed: 2020-4-28.

[54] 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.

[55] 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.

[56] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[57] Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 267–282. Springer, 2012. doi: 10.1007/978-3-642-27940-9_18.

[58] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008. doi: 10.1007/978-3-540-70545-1_40.

[59] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 214–228. Springer, 2009. doi: 10.1007/978-3-540-93900-9_19.

[60] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. doi: 10.1145/360248.360252.

[61] 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. doi: 10.1145/1629575.1629596.

[62] 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. doi: 10.1007/978-3-642-54862-8_26.

[63] Leslie Lamport. The TLA+ hyperbook. https://lamport.azurewebsites.net/tla/hyperbook.html. Accessed: 2020-4-28.

[64] Lem Repository. Lem project. https://github.com/rems-project/lem. Accessed: 2019-5-31.

[65] Xavier Leroy. The CompCert C verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[66] 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.

[67] Wenjie Li, Dongpeng Xu, Wei Wu, Xiaorui Gong, Xiaobo Xiang, Yan Wang, Qianxiang Zeng, et al. Memory access integrity: detecting fine-grained memory access errors in binary code. *Cybersecurity*, 2(1):1–18, 2019. doi: 10.1186/s42400-019-0035-x.

[68] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, 2006. doi: 10.1145/1181309.1181314.

[69] 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.

[70] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016. doi: 10.1145/2931037.2931047.

[71] César A Munoz. Batch proving and proof scripting in PVS. Technical Report NIA Report No. 2007-03, 2007.

[72] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic – Improved. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 78–81. IEEE, 2012.

[73] Anthony Narkawicz, César A Munoz, and Aaron M Dutle. The MINERVA software development process. In *NASA Formal Methods Symposium (NFM) 2017*, number NF1676L-26800, 2017. doi: 10.29007/5jlw.

[74] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002. ISBN 978-3-540-45949-1.

[75] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches. *arXiv preprint arXiv:1702.00719*, 2017. doi: 10.48550/arXiv.1702.00719.

[76] OPEV Repository. OPEV bug report. https://github.com/ssrg-vt/OPEV/blob/master/BugReport.pdf. Accessed: 2022-4-28.

[77] 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. doi: 10.1007/BFb0031813.

[78] Sam Owre. Random testing in PVS. In *Workshop on automated formal methods (AFM)*, volume 10, 2006.

[79] Sam Owre, John M Rushby, and Natarajan 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, June 1992. Springer. doi: 10.1007/3-540-55602-8_217.

[80] 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. doi: 10.1145/1831708.1831741.

[81] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851. IEEE, 2021. doi: 10.48550/arXiv.2007.14266.

[82] Jungmin Park, Xiaolin Xu, Yier Jin, Domenic Forte, and Mark Tehranipoor. Power-based side-channel instruction-level disassembler. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018. doi: 10.1109/DAC.2018.8465848.

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

[84] PVS Snapshots. `http://www.csl.sri.com/users/owre/drop/pvs-snapshots/`. Accessed: 2022-4-28.

[85] Radare2 Repository. Radare2: Unix-like reverse engineering framework. `https://github.com/radareorg/radare2`. Accessed: 2022-4-28.

[86] Alastair Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168. IEEE, 2016. doi: 10.1109/FMCAD.2016.7886675.

[87] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. doi: 10.1090/S0002-9947-1953-0053041-6.

[88] 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. doi: 10.1145/2815400.2815411.

[89] Roman Rohleder. Hands-On Ghidra - A tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 77–78, 2019. doi: 10.1145/3338503.3357725.

[90] John Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998. doi: 10.1109/32.713327.

[91] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *The Network and Distributed System Security (NDSS) Symposium*, volume 4, pages 159–169, 2004.

[92] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[93] Sail Repository. The Sail ISA specification language. https://github.com/rems-project/sail. Accessed: 2019-5-31.

[94] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236, 2009.

[95] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54. IEEE, 2002. doi: 10.1109/WCRE.2002.1173063.

[96] seL4 Repository. The seL4 microkernel. https://github.com/seL4/seL4. Accessed: 2022-4-28.

[97] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, volume 4144, pages 419–423. Springer, 2006. doi: 10.1007/11817963_38.

[98] 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. doi: 10.1145/1095430.1081750.

[99] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[100] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 471–482. ACM, 2013. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462183.

[101] Rebecca Shapiro, Sergey Bratus, and Sean W Smith. “Weird Machines” in ELF: A spotlight on the underappreciated metadata. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, 2013.

[102] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SoK: (State of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016. doi: 10.1109/SP.2016.17.

[103] Michael Steil. 17 mistakes Microsoft made in the Xbox security system. In *22nd Chaos Communication Congress*, 2005.

[104] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 132–143, 1977. doi: 10.1145/512950.512963.

[105] Chris Swinchatt. MLDisasm. https://github.com/ChrisSwinchatt/MLDisasm. Accessed: 2022-4-28.

[106] Amer Tahat, Sarang Joshi, Pronnoy Goswami, and Binoy Ravindran. Scalable translation validation of unverified legacy OS code. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE, 2019. doi: 10.23919/FMCAD.2019.8894252.

[107] Éric Tanter and Nicolas Tabareau. Gradual certified programming in Coq. In *Proceedings of the 11th Symposium on Dynamic Languages*, pages 26–40, 2015. doi: 10.1145/2816707.2816710.

[108] Janet M Twomey and Alice E Smith. Validation and verification, 1997.

[109] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *2013 IEEE Symposium on Security and Privacy*, pages 430–444. IEEE, 2013. doi: 10.1109/SP.2013.36.

[110] Muralidaran Vijayaraghavan, Adam Chlipala, Nirav Dave, et al. Modular deductive verification of multiprocessor hardware designs. In *Computer Aided Verification*, pages 109–127. Springer, 2015. ISBN 978-3-319-21667-6.

[111] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated software engineering*, 10(2):203–232, 2003. doi: 10.1023/A:1022920129859.

[112] Yusuke Wada and Shigeru Kusakabe. Performance evaluation of a testing framework using QuickCheck and Hadoop. *Information and Media Technologies*, 7(2):694–700, 2012.

[113] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *The Network and Distributed System Security (NDSS) Symposium*, 2017. doi: 10.14722/NDSS.2017.23225.

[114] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011. doi: 10.1007/978-3-642-23808-6_34.

[115] Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 273–285. Springer, 2014.

[116] 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. doi: 10.1145/2737924.2737958.

[117] Zhichen Xu, Barton P Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 70–82, 2000. doi: 10.1145/349299.349313.

[118] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999. doi: 10.1007/3-540-48153-2_6.

[119] 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. doi: 10.1145/2576195.2576208.

[120] Weixiong Zhang. *State-space search: Algorithms, complexity, extensions, and applications.* Springer Science & Business Media, 1999. ISBN 978-1-4612-1538-7.

Appendices

Appendix A

DSV Execution Results

Table A.1: Execution results for Coreutils library on 8 different disassemblers.

		# of total	# of white	# of grey	# of black	Ratio of grey vs. white	# of indirects	Missing instr	Sound
objdump	[	6831	5554	55	1222	0.01	96		
	b2sum	7662	5828	271	1563	0.05	103		
	base32	4464	3267	63	1134	0.02	83		
	base64	4406	3206	62	1138	0.02	83		
	basename	3310	2217	18	1075	0.01	59		
	basenc	5892	2628	290	2974	0.11	65		
	cat	3915	2734	61	1120	0.02	70		
	chcon	8305	5777	87	2441	0.02	147		
	chgrp	8793	6214	112	2467	0.02	150		
	chmod	8429	5919	101	2409	0.02	142		
	chown	9141	6548	100	2493	0.02	154		
	chroot	4457	3336	61	1060	0.02	83		
	cksum	3445	2275	72	1098	0.03	70		
	comm	4351	3115	61	1175	0.02	86		
	cp	15805	11551	254	4000	0.02	244		
	csplit	20111	9033	229	10849	0.03	150		
	cut	4750	3677	28	1045	0.01	91		
	date	14454	12122	421	1911	0.03	151		
	dd	10571	7993	377	2201	0.05	132		
	df	13036	9468	242	3326	0.03	171		
	dir	19029	12751	417	5861	0.03	230		
	dircolors	4223	3063	28	1132	0.01	86		
	dirname	3238	765	8	2465	0.01	28		
	du	29764	19356	755	9653	0.04	326		
	echo	3365	929	13	2423	0.01	34		
	env	5037	3870	44	1123	0.01	86		
	expand	3928	2742	112	1074	0.04	79		
	expr	19538	16778	508	2252	0.03	246		
	factor	10530	8596	71	1863	0.01	152		
	false	3016	610	8	2398	0.01	23		
	fmt	4723	3686	31	1006	0.01	91		

objdump	fold	3797	2786	38	973	0.01	78
	getlimits	4150	1600	36	2514	0.02	38
	ginstall	19407	15318	425	3664	0.03	299
	groups	3597	2455	19	1123	0.01	65
	head	5086	3947	41	1098	0.01	82
	hostid	3170	1987	30	1153	0.02	55
	id	4679	3364	110	1205	0.03	76
	join	5789	3123	250	2416	0.08	76
	kill	3738	2613	40	1085	0.02	67
	link	3229	2045	38	1146	0.02	57
	ln	9351	5812	240	3299	0.04	154
	logname	3191	2004	30	1157	0.01	56
	ls	19029	12751	417	5861	0.03	230
	make-prime-list	530	406	7	117	0.02	18
	md5sum	5573	4256	76	1241	0.02	88
	mkdir	6897	5289	147	1461	0.03	107
	mkfifo	3575	2389	63	1123	0.03	61
	mknod	4101	2775	216	1110	0.08	65
	mktemp	4910	3551	74	1285	0.02	107
	mv	18420	14398	196	3826	0.01	294
	nice	3649	2511	27	1111	0.01	58
	nl	18435	15959	492	1984	0.03	227
	nohup	3881	2529	94	1258	0.04	73
	nproc	3663	2499	61	1103	0.02	66
	numfmt	7625	6003	219	1403	0.04	94
	od	9185	4379	142	4664	0.03	93
	paste	3777	2766	20	991	0.01	71
	pathchk	3424	2402	28	994	0.01	59
	pinky	4220	1820	26	2374	0.01	68
	pr	10087	8397	119	1571	0.01	158
	printenv	3126	718	8	2400	0.01	23
	printf	6800	5426	64	1310	0.01	99
	ptx	24419	20905	799	2715	0.04	265
	pwd	3659	2573	27	1059	0.01	72
	readlink	5413	2263	106	3044	0.05	59
	realpath	5828	2644	89	3095	0.03	72
	rm	9137	6288	103	2746	0.02	153
	rmdir	5406	4164	75	1167	0.02	80
	runcon	3171	768	8	2395	0.01	23
	seq	6482	4836	101	1545	0.02	84
	shred	7665	5746	264	1655	0.05	133
	shuf	7629	5516	342	1771	0.06	135
	sleep	3553	2325	45	1183	0.02	69
	split	6882	5416	102	1364	0.02	117
	stat	11646	9765	115	1766	0.01	173

objdump	stdbuf	6121	4786	58	1277	0.01	96
	stty	8644	7062	111	1471	0.02	101
	sum	4906	2810	178	1918	0.06	80
	sync	3404	2258	58	1088	0.03	60
	tac	18421	15823	463	2135	0.03	228
	tail	9660	5940	97	3623	0.02	117
	tee	3908	2671	54	1183	0.02	78
	test	6449	4976	36	1437	0.01	86
	timeout	4249	2945	33	1271	0.01	93
	touch	12454	10276	235	1943	0.02	140
	tr	5525	4129	187	1209	0.05	91
	true	3017	610	8	2399	0.01	23
	truncate	4110	3002	27	1081	0.01	65
	tsort	4201	2812	57	1332	0.02	85
	tty	3079	1990	18	1071	0.01	54
	uname	3251	2168	12	1071	0.01	54
	unexpand	4009	2900	46	1063	0.02	83
	uniq	5026	3779	35	1212	0.01	102
	unlink	3210	2049	36	1125	0.02	56
	uptime	6080	4448	69	1563	0.02	93
	users	3483	2269	43	1171	0.02	65
	vdir	19029	12751	417	5861	0.03	230
	wc	5262	3243	93	1926	0.03	86
	who	6493	5237	62	1194	0.01	92
	whoami	3204	2016	30	1158	0.01	57
	yes	3351	837	22	2492	0.03	34
radare2	[	6930	5554	54	1322	0.01	96
	b2sum	7761	5828	269	1664	0.05	103
	base32	4563	3267	61	1235	0.02	83
	base64	4505	3206	61	1238	0.02	83
	basename	3409	2217	18	1174	0.01	59
	basenc	5991	2628	289	3074	0.11	65
	cat	4014	2734	59	1221	0.02	70
	chcon	8398	5776	83	2539	0.01	146
	chgrp	8886	6213	106	2567	0.02	149
	chmod	8522	5918	97	2507	0.02	141
	chown	9234	6547	93	2594	0.01	153
	chroot	4556	3336	60	1160	0.02	83
	cksum	3544	2275	70	1199	0.03	70
	comm	4450	3115	64	1271	0.02	86
	cp	15898	11550	248	4100	0.02	243
	csplit	20210	9033	226	10951	0.03	150
	cut	4849	3677	27	1145	0.01	91
	date	14553	12122	414	2017	0.03	151
	dd	10670	7993	322	2355	0.04	132

radare2	df	13135	9468	239	3428	0.03	171		
	dir	19124	12900	320	5904	0.02	231	×	×
	dircolors	4318	3064	24	1230	0.01	84	×	×
	dirname	3337	765	8	2564	0.01	28		
	du	29857	19355	747	9755	0.04	325		
	echo	3464	929	13	2522	0.01	34		
	env	5136	3870	43	1223	0.01	86		
	expand	4027	2742	111	1174	0.04	79		
	expr	19637	16778	504	2355	0.03	246		
	factor	10629	8596	67	1966	0.01	152		
	false	3115	610	8	2497	0.01	23		
	fmt	4822	3686	29	1107	0.01	91		
	fold	3896	2786	37	1073	0.01	78		
	getlimits	4249	1600	35	2614	0.02	38		
	ginstall	19500	15317	442	3741	0.03	298		
	groups	3696	2455	23	1218	0.01	65		
	head	5185	3947	39	1199	0.01	82		
	hostid	3269	1987	30	1252	0.02	55		
	id	4778	3364	110	1304	0.03	76		
	join	5888	3123	248	2517	0.08	76		
	kill	3837	2613	40	1184	0.02	67		
	link	3328	2045	37	1246	0.02	57		
	ln	9450	5812	236	3402	0.04	154		
	logname	3290	2004	30	1256	0.01	56		
	ls	19124	12900	320	5904	0.02	231	×	×
	make-prime-list	627	406	55	166	0.14	18		
	md5sum	5672	4256	75	1341	0.02	88		
	mkdir	6996	5289	145	1562	0.03	107		
	mkfifo	3674	2389	62	1223	0.03	61		
	mknod	4200	2775	214	1211	0.08	65		
	mktemp	5009	3551	74	1384	0.02	107		
	mv	18513	14397	190	3926	0.01	293		
	nice	3748	2511	27	1210	0.01	58		
	nl	18534	15959	489	2086	0.03	227		
	nohup	3980	2529	92	1359	0.04	73		
	nproc	3762	2499	61	1202	0.02	66		
	numfmt	7724	6003	218	1503	0.04	94		
	od	9284	4379	140	4765	0.03	93		
	paste	3876	2766	19	1091	0.01	71		
	pathchk	3523	2402	26	1095	0.01	59		
	pinky	4319	1820	25	2474	0.01	68		
	pr	10186	8397	117	1672	0.01	158		
	printenv	3225	718	8	2499	0.01	23		
	printf	6899	5426	63	1410	0.01	99		
	ptx	24518	20905	793	2820	0.04	265		

radare2	pwd	3758	2573	26	1159	0.01	72		
	readlink	5512	2263	103	3146	0.05	59		
	realpath	5927	2644	86	3197	0.03	72		
	rm	9230	6287	99	2844	0.02	152		
	rmdir	5505	4164	74	1267	0.02	80		
	runcon	3270	768	8	2494	0.01	23		
	seq	6581	4836	100	1645	0.02	84		
	shred	7764	5746	261	1757	0.05	133		
	shuf	7722	5515	340	1867	0.06	134		
	sleep	3652	2325	45	1282	0.02	69		
	split	6981	5416	99	1466	0.02	117		
	stat	11745	9765	111	1869	0.01	173		
	stdbuf	6220	4786	57	1377	0.01	96		
	stty	8743	7062	109	1572	0.02	101		
	sum	5005	2810	177	2018	0.06	80		
	sync	3503	2258	56	1189	0.02	60		
	tac	18520	15823	459	2238	0.03	228		
	tail	9759	5940	92	3727	0.02	117		
	tee	4007	2671	52	1284	0.02	78		
	test	6548	4976	35	1537	0.01	86		
	timeout	4348	2945	33	1370	0.01	93		
	touch	12553	10276	229	2048	0.02	140		
	tr	5624	4129	186	1309	0.05	91		
	true	3116	610	8	2498	0.01	23		
	truncate	4209	3002	26	1181	0.01	65		
	tsort	4300	2812	55	1433	0.02	85		
	tty	3178	1990	18	1170	0.01	54		
	uname	3350	2168	12	1170	0.01	54		
	unexpand	4108	2900	45	1163	0.02	83		
	uniq	5125	3779	32	1314	0.01	102		
	unlink	3309	2049	35	1225	0.02	56		
	uptime	6179	4448	69	1662	0.02	93		
	users	3582	2269	43	1270	0.02	65		
	vdir	19124	12900	320	5904	0.02	231	×	×
	wc	5357	3244	90	2023	0.03	84	×	×
	who	6592	5237	61	1294	0.01	92		
	whoami	3303	2016	30	1257	0.01	57		
	yes	3450	837	22	2591	0.03	34		
angr	[	6936	5554	54	1328	0.01	96		
	b2sum	7767	5828	269	1670	0.05	103		
	base32	4569	3267	61	1241	0.02	83		
	base64	4511	3206	61	1244	0.02	83		
	basename	3415	2217	18	1180	0.01	59		
	basenc	5997	2628	289	3080	0.11	65		
	cat	4020	2734	59	1227	0.02	70		

angr	chcon	8410	5777	83	2550	0.01	147
	chgrp	8898	6214	106	2578	0.02	150
	chmod	8534	5919	97	2518	0.02	142
	chown	9246	6548	93	2605	0.01	154
	chroot	4562	3336	60	1166	0.02	83
	cksum	3550	2275	70	1205	0.03	70
	comm	4456	3115	64	1277	0.02	86
	cp	15910	11551	248	4111	0.02	244
	csplit	20216	9033	226	10957	0.03	150
	cut	4855	3677	27	1151	0.01	91
	date	14559	12122	414	2023	0.03	151
	dd	10676	7993	373	2310	0.05	132
	df	13141	9468	239	3434	0.03	171
	dir	19134	12751	413	5970	0.03	230
	dircolors	4328	3063	26	1239	0.01	86
	dirname	3343	765	8	2570	0.01	28
	du	29869	19356	747	9766	0.04	326
	echo	3470	929	13	2528	0.01	34
	env	5142	3870	43	1229	0.01	86
	expand	4033	2742	111	1180	0.04	79
	expr	19643	16778	504	2361	0.03	246
	factor	10635	8596	67	1972	0.01	152
	false	3121	610	8	2503	0.01	23
	fmt	4828	3686	29	1113	0.01	91
	fold	3902	2786	37	1079	0.01	78
	getlimits	4255	1600	36	2619	0.02	38
	ginstall	19512	15318	442	3752	0.03	299
	groups	3702	2455	23	1224	0.01	65
	head	5191	3947	39	1205	0.01	82
	hostid	3275	1987	30	1258	0.02	55
	id	4784	3364	110	1310	0.03	76
	join	5894	3123	248	2523	0.08	76
	kill	3843	2613	40	1190	0.02	67
	link	3334	2045	37	1252	0.02	57
	ln	9456	5812	236	3408	0.04	154
	logname	3296	2004	30	1262	0.01	56
	ls	19134	12751	413	5970	0.03	230
	make-prime-list	633	406	55	172	0.14	18
	md5sum	5678	4256	75	1347	0.02	88
	mkdir	7002	5289	145	1568	0.03	107
	mkfifo	3680	2389	62	1229	0.03	61
	mknod	4206	2775	214	1217	0.08	65
	mktemp	5015	3551	74	1390	0.02	107
	mv	18525	14398	190	3937	0.01	294
	nice	3754	2511	27	1216	0.01	58

angr	nl	18540	15959	489	2092	0.03	227
	nohup	3986	2529	92	1365	0.04	73
	nproc	3768	2499	61	1208	0.02	66
	numfmt	7730	6003	218	1509	0.04	94
	od	9290	4379	140	4771	0.03	93
	paste	3882	2766	19	1097	0.01	71
	pathchk	3529	2402	26	1101	0.01	59
	pinky	4325	1820	25	2480	0.01	68
	pr	10192	8397	117	1678	0.01	158
	printenv	3231	718	8	2505	0.01	23
	printf	6905	5426	63	1416	0.01	99
	ptx	24524	20905	793	2826	0.04	265
	pwd	3764	2573	26	1165	0.01	72
	readlink	5518	2263	103	3152	0.05	59
	realpath	5933	2644	86	3203	0.03	72
	rm	9242	6288	99	2855	0.02	153
	rmdir	5511	4164	74	1273	0.02	80
	runcon	3276	768	8	2500	0.01	23
	seq	6587	4836	100	1651	0.02	84
	shred	7770	5746	261	1763	0.05	133
	shuf	7734	5516	340	1878	0.06	135
	sleep	3658	2325	45	1288	0.02	69
	split	6987	5416	99	1472	0.02	117
	stat	11751	9765	111	1875	0.01	173
	stdbuf	6226	4786	57	1383	0.01	96
	stty	8749	7062	109	1578	0.02	101
	sum	5011	2810	177	2024	0.06	80
	sync	3509	2258	56	1195	0.02	60
	tac	18526	15823	459	2244	0.03	228
	tail	9765	5940	92	3733	0.02	117
	tee	4013	2671	52	1290	0.02	78
	test	6554	4976	35	1543	0.01	86
	timeout	4354	2945	33	1376	0.01	93
	touch	12559	10276	229	2054	0.02	140
	tr	5630	4129	186	1315	0.05	91
	true	3122	610	8	2504	0.01	23
	truncate	4215	3002	26	1187	0.01	65
	tsort	4306	2812	55	1439	0.02	85
	tty	3184	1990	18	1176	0.01	54
	uname	3356	2168	12	1176	0.01	54
	unexpand	4114	2900	45	1169	0.02	83
	uniq	5131	3779	32	1320	0.01	102
	unlink	3315	2049	35	1231	0.02	56
	uptime	6185	4448	69	1668	0.02	93
	users	3588	2269	43	1276	0.02	65

angr	vdir	19134	12751	413	5970	0.03	230		
	wc	5367	3243	92	2032	0.03	86		
	who	6598	5237	61	1300	0.01	92		
	whoami	3309	2016	30	1263	0.01	57		
	yes	3456	837	22	2597	0.03	34		
BAP	[	10264	2691	356	7217	0.13	63	×	
	b2sum	14190	4193	223	9774	0.05	82	×	
	base32	7258	1725	177	5356	0.1	61	×	
	base64	7731	1664	177	5890	0.11	61	×	
	basename	5894	826	114	4954	0.14	37	×	
	basenc	7631	916	142	6573	0.16	34	×	
	cat	6875	1362	117	5396	0.09	47	×	
	chcon	12773	4080	182	8511	0.04	121	×	
	chgrp	14333	4420	139	9774	0.03	127	×	
	chmod	14887	4255	188	10444	0.04	118	×	
	chown	14455	4739	214	9502	0.05	131	×	
	chroot	7584	1370	204	6010	0.15	55	×	
	cksum	6386	930	52	5404	0.06	45	×	
	comm	8626	1713	139	6774	0.08	66	×	
	cp	30012	8460	561	20991	0.07	208	×	
	csplit	29985	4668	235	25082	0.05	113	×	
	cut	8078	2257	120	5701	0.05	71	×	
	date	17099	5451	1144	10504	0.21	114	×	
	dd	19694	5461	502	13731	0.09	109	×	×
	df	21992	2219	304	19469	0.14	67	×	
	dir	28906	5718	667	22521	0.12	150	×	×
	dircolors	5608	1634	120	3854	0.07	64	×	
	dirname	5654	584	32	5038	0.05	26	×	
	du	46684	14645	1074	30965	0.07	287	×	×
	echo	5539	653	39	4847	0.06	23	×	
	env	8105	2344	129	5632	0.06	66	×	
	expand	7373	1320	205	5848	0.16	56	×	
	expr	29179	12708	578	15893	0.05	213	×	
	factor	16636	7299	164	9173	0.02	142	×	
	false	4965	429	32	4504	0.07	21	×	
	fmt	7744	2138	142	5464	0.07	67	×	
	fold	6526	1249	152	5125	0.12	55	×	
	getlimits	7160	1419	59	5682	0.04	36	×	
	ginstall	39673	10716	302	28655	0.03	252	×	
	groups	6360	1050	119	5191	0.11	42	×	
	head	8553	2226	158	6169	0.07	60	×	
	hostid	5418	591	126	4701	0.21	30	×	
	id	8526	1885	215	6426	0.11	59	×	
	join	10113	1538	249	8326	0.16	55	×	
	kill	6308	1209	139	4960	0.11	47	×	

BAP	link	5572	663	122	4787	0.18	33	×	
	ln	18841	4127	313	14401	0.08	130	×	
	logname	5418	609	126	4683	0.21	32	×	
	ls	28906	5718	667	22521	0.12	150	×	×
	make-prime-list	1167	406	52	709	0.13	18		
	md5sum	8915	2779	154	5982	0.06	66	×	
	mkdir	11711	1672	183	9856	0.11	62	×	
	mkfifo	6713	1036	52	5625	0.05	39	×	
	mknod	7022	1282	162	5578	0.13	43	×	
	mktemp	9206	2103	172	6931	0.08	87	×	
	mv	36229	11200	290	24739	0.03	261	×	
	nice	5788	875	123	4790	0.14	34	×	
	nl	25960	4941	196	20823	0.04	128	×	
	nohup	7358	492	40	6826	0.08	23	×	
	nproc	6717	878	154	5685	0.18	39	×	
	numfmt	12126	3433	187	8506	0.05	70	×	×
	od	14225	1494	485	12246	0.32	59	×	
	paste	6474	1313	120	5041	0.09	49	×	
	pathchk	6123	965	118	5040	0.12	36	×	
	pinky	7019	841	47	6131	0.06	37	×	
	pr	14900	1824	792	12284	0.43	79	×	
	printenv	5188	535	32	4621	0.06	21	×	
	printf	11414	1549	162	9703	0.1	59	×	
	ptx	38123	6170	221	31732	0.04	115	×	
	pwd	6222	1058	118	5046	0.11	51	×	
	readlink	10410	1251	91	9068	0.07	46	×	
	realpath	11368	1251	108	10009	0.09	46	×	
	rm	14956	1768	205	12983	0.12	69	×	
	rmdir	8230	765	114	7351	0.15	36	×	
	runcon	5375	585	32	4758	0.05	21	×	
	seq	11008	969	253	9786	0.26	42	×	
	shred	14080	1616	538	11926	0.33	67	×	
	shuf	13855	1849	340	11666	0.18	74	×	
	sleep	7078	716	44	6318	0.06	38	×	
	split	9845	2021	300	7524	0.15	66	×	
	stat	15345	1291	129	13925	0.1	48	×	
	stdbuf	10669	1360	172	9137	0.13	34	×	
	stty	10982	2957	246	7779	0.08	55	×	
	sum	8562	1277	220	7065	0.17	52	×	×
	sync	5758	768	129	4861	0.17	31	×	
	tac	27009	3910	154	22945	0.04	86	×	
	tail	17006	3914	167	12925	0.04	94	×	
	tee	7402	1196	129	6077	0.11	56	×	
	test	9159	1937	377	6845	0.19	42	×	
	timeout	6389	1038	118	5233	0.11	47	×	

BAP	touch	14986	3222	190	11574	0.06	76	×	
	tr	8997	1218	189	7590	0.16	41	×	
	true	4970	428	32	4510	0.07	21	×	
	truncate	6430	1078	149	5203	0.14	40	×	
	tsort	7920	978	128	6814	0.13	46	×	
	tty	4981	594	114	4273	0.19	29	×	
	uname	5652	769	108	4775	0.14	30	×	
	unexpand	7445	1065	152	6228	0.14	44	×	
	uniq	8375	1385	138	6852	0.1	53	×	
	unlink	5504	461	40	5003	0.09	22	×	
	uptime	8375	482	40	7853	0.08	22	×	
	users	6092	463	40	5589	0.09	22	×	
	vdir	28906	848	469	27589	0.55	41	×	
	wc	9968	1859	166	7943	0.09	74	×	
	who	10487	895	114	9478	0.13	34	×	
	whoami	5463	460	40	4963	0.09	22	×	
	yes	6156	465	40	5651	0.09	22	×	
Ghidra	[	6666	5554	54	1058	0.01	96		
	b2sum	7528	5828	108	1592	0.02	103		
	base32	4378	3267	60	1051	0.02	83		
	base64	4317	3206	60	1051	0.02	83		
	basename	3256	2217	18	1021	0.01	59		
	basenc	5715	2628	50	3037	0.02	65		
	cat	3850	2734	59	1057	0.02	70		
	chcon	7817	5564	78	2175	0.01	143	×	
	chgrp	8483	6214	106	2163	0.02	150		
	chmod	8055	5819	97	2139	0.02	140	×	
	chown	8808	6548	93	2167	0.01	154		
	chroot	4382	3336	60	986	0.02	83		
	cksum	3394	2275	70	1049	0.03	70		
	comm	4258	3115	49	1094	0.02	86		
	cp	15179	11551	249	3379	0.02	244		
	csplit	19529	9033	130	10366	0.01	150		×
	cut	4646	3677	27	942	0.01	91		
	date	14178	12122	426	1630	0.04	151		
	dd	10337	7993	373	1971	0.05	132		
	df	12661	9468	224	2969	0.02	171		
	dir	18303	12751	267	5285	0.02	230		×
	dircolors	4133	3063	26	1044	0.01	86		
	dirname	3191	765	8	2418	0.01	28		
	du	28779	19356	564	8859	0.03	326		×
	echo	3313	929	13	2371	0.01	34		
	env	4956	3870	43	1043	0.01	86		
	expand	3826	2742	99	985	0.04	79		
	expr	18952	16778	484	1690	0.03	246		×

Ghidra	factor	10249	8596	67	1586	0.01	152	
	false	2977	610	8	2359	0.01	23	
	fmt	4619	3686	29	904	0.01	91	
	fold	3727	2786	37	904	0.01	78	
	getlimits	4101	1600	36	2465	0.02	38	
	ginstall	18796	15318	296	3182	0.02	299	
	groups	3541	2455	23	1063	0.01	65	
	head	5001	3947	39	1015	0.01	82	
	hostid	3129	1987	30	1112	0.02	55	
	id	4583	3364	121	1098	0.04	76	
	join	5601	3123	135	2343	0.04	76	
	kill	3684	2613	40	1031	0.02	67	
	link	3185	2045	37	1103	0.02	57	
	ln	9049	5812	221	3016	0.04	154	
	logname	3149	2004	30	1115	0.01	56	
	ls	18303	12751	267	5285	0.02	230	
	make-prime-list	606	406	52	148	0.13	18	
	md5sum	5471	4256	60	1155	0.01	88	
	mkdir	6708	5289	121	1298	0.02	107	
	mkfifo	3511	2389	62	1060	0.03	61	
	mknod	4029	2775	68	1186	0.02	65	
	mktemp	4806	3551	74	1181	0.02	107	
	mv	17771	14398	190	3183	0.01	294	
	nice	3581	2511	27	1043	0.01	58	
	nl	17881	15959	462	1460	0.03	227	×
	nohup	3824	2529	92	1203	0.04	73	
	nproc	3604	2499	45	1060	0.02	66	×
	numfmt	7452	6003	113	1336	0.02	94	
	od	8941	4379	145	4417	0.03	93	
	paste	3709	2766	19	924	0.01	71	
	pathchk	3380	2402	26	952	0.01	59	
	pinky	4144	1820	25	2299	0.01	68	
	pr	9769	8397	103	1269	0.01	158	
	printenv	3088	718	8	2362	0.01	23	
	printf	6649	5426	63	1160	0.01	99	×
	ptx	23675	20905	555	2215	0.03	265	×
	pwd	3615	2573	26	1016	0.01	72	
	readlink	5263	2263	103	2897	0.05	59	
	realpath	5658	2644	86	2928	0.03	72	
	rm	8800	6288	99	2413	0.02	153	
	rmdir	5293	4164	74	1055	0.02	80	
	runcon	3133	768	8	2357	0.01	23	
	seq	6368	4836	100	1432	0.02	84	
	shred	7475	5746	151	1578	0.03	133	
	shuf	7419	5516	179	1724	0.03	135	

Ghidra	sleep	3495	2325	45	1125	0.02	69	
	split	6755	5416	83	1256	0.02	117	
	stat	11321	9765	96	1460	0.01	173	
	stdbuf	5997	4786	57	1154	0.01	96	×
	stty	8450	7062	109	1279	0.02	101	
	sum	4797	2810	128	1859	0.05	80	
	sync	3360	2258	56	1046	0.02	60	
	tac	17903	15823	444	1636	0.03	228	×
	tail	9297	5940	94	3263	0.02	117	
	tee	3834	2671	37	1126	0.01	78	
	test	6298	4976	35	1287	0.01	86	
	timeout	4175	2945	33	1197	0.01	93	
	touch	12206	10276	229	1701	0.02	140	
	tr	5414	4129	69	1216	0.02	91	
	true	2978	610	8	2360	0.01	23	
	truncate	4033	3002	26	1005	0.01	65	
	tsort	4118	2812	55	1251	0.02	85	
	tty	3039	1990	18	1031	0.01	54	
	uname	3210	2168	12	1030	0.01	54	
	unexpand	3906	2900	45	961	0.02	83	
	uniq	4915	3779	32	1104	0.01	102	
	unlink	3168	2049	35	1084	0.02	56	
	uptime	5975	4448	69	1458	0.02	93	
	users	3437	2269	43	1125	0.02	65	
	vdir	18303	12751	267	5285	0.02	230	×
	wc	5130	3243	92	1795	0.03	86	
	who	6369	5237	61	1071	0.01	92	
	whoami	3162	2016	30	1116	0.01	57	
	yes	3300	837	22	2441	0.03	34	
Dyninst	[	6764	5517	66	1181	0.01	97	×
	b2sum	7573	5754	254	1565	0.04	103	×
	base32	4407	3257	58	1092	0.02	83	×
	base64	4348	3191	62	1095	0.02	83	×
	basename	3269	2222	16	1031	0.01	60	×
	basenc	5777	2622	288	2867	0.11	65	×
	cat	3865	2669	80	1116	0.03	67	×
	chcon	8144	5622	83	2439	0.01	146	×
	chgrp	8640	6079	80	2481	0.01	149	×
	chmod	8289	5741	89	2459	0.02	141	×
	chown	8983	6223	93	2667	0.01	152	×
	chroot	4395	3151	75	1169	0.02	82	×
	cksum	3398	2262	60	1076	0.03	70	×
	comm	4287	3053	63	1171	0.02	86	×
	cp	15503	10771	163	4569	0.02	234	×
	csplit	19899	9477	191	10231	0.02	156	×

Dyninst	cut	4686	3672	19	995	0.01	91	×
	date	14333	10789	1761	1783	0.16	145	×
	dd	10451	7860	381	2210	0.05	133	×
	df	12827	9398	223	3206	0.02	168	×
	dir	18694	12845	329	5520	0.03	230	×
	dircolors	4162	3045	24	1093	0.01	86	×
	dirname	3201	765	8	2428	0.01	28	×
	du	29379	19223	756	9400	0.04	321	×
	echo	3330	929	13	2388	0.01	34	×
	env	4975	3850	43	1082	0.01	86	×
	expand	3874	2707	123	1044	0.05	79	×
	expr	19339	16775	449	2115	0.03	246	×
	factor	10384	6088	35	4261	0.01	123	×
	false	2984	610	8	2366	0.01	23	×
	fmt	4663	3647	52	964	0.01	90	×
	fold	3746	2775	36	935	0.01	78	×
	getlimits	4103	1295	37	2771	0.03	29	×
	ginstall	19058	12227	461	6370	0.04	274	×
	groups	3546	2461	20	1065	0.01	66	×
	head	5027	3792	38	1197	0.01	81	×
	hostid	3136	1990	28	1118	0.01	56	×
	id	4618	3404	95	1119	0.03	79	×
	join	5706	2801	277	2628	0.1	61	×
	kill	3691	2159	22	1510	0.01	54	×
	link	3193	2068	27	1098	0.01	57	×
	ln	9151	5663	226	3262	0.04	152	×
	logname	3156	2007	28	1121	0.01	57	×
	ls	18694	12845	329	5520	0.03	230	×
	make-prime-list	600	406	55	139	0.14	18	×
	md5sum	5506	4167	74	1265	0.02	88	×
	mkdir	6809	3397	134	3278	0.04	94	×
	mkfifo	3537	2436	30	1071	0.01	61	×
	mknod	4058	2747	214	1097	0.08	64	×
	mktemp	4822	3553	73	1196	0.02	108	×
	mv	18066	10297	162	7607	0.02	228	×
	nice	3611	2514	25	1072	0.01	59	×
	nl	18256	15924	486	1846	0.03	228	×
	nohup	3832	2536	60	1236	0.02	73	×
	nproc	3617	2502	59	1056	0.02	67	×
	numfmt	7554	6061	155	1338	0.03	94	×
	od	9069	4336	136	4597	0.03	92	×
	paste	3733	2723	18	992	0.01	71	×
	pathchk	3387	2306	25	1056	0.01	56	×
	pinky	4157	1822	37	2298	0.02	69	×
	pr	9953	7891	462	1600	0.06	153	×

Dyninst	printenv	3094	718	8	2368	0.01	23	×
	printf	6729	5363	82	1284	0.02	98	×
	ptx	24187	20967	714	2506	0.03	263	×
	pwd	3606	2546	24	1036	0.01	72	×
	readlink	5315	2270	93	2952	0.04	59	×
	realpath	5724	2609	85	3030	0.03	71	×
	rm	8977	5951	88	2938	0.01	152	×
	rmdir	5352	2230	19	3103	0.01	65	×
	runcon	3140	768	8	2364	0.01	23	×
	seq	6426	4900	121	1405	0.02	84	×
	shred	7545	3254	91	4200	0.03	89	×
	shuf	7488	4869	336	2283	0.07	123	×
	sleep	3507	2327	43	1137	0.02	69	×
	split	6798	5183	107	1508	0.02	116	×
	stat	11494	2742	20	8732	0.01	67	×
	stdbuf	6058	4597	70	1391	0.02	79	×
	stty	8572	6962	117	1493	0.02	100	×
	sum	4840	2757	166	1917	0.06	79	×
	sync	3366	2274	24	1068	0.01	60	×
	tac	18238	8921	136	9181	0.02	166	×
	tail	9522	6068	193	3261	0.03	123	×
	tee	3850	2630	52	1168	0.02	77	×
	test	6385	4939	47	1399	0.01	87	×
	timeout	4173	2947	31	1195	0.01	93	×
	touch	12334	8743	1847	1744	0.21	134	×
	tr	5462	4132	184	1146	0.04	92	×
	true	2984	610	8	2366	0.01	23	×
	truncate	4065	2887	25	1153	0.01	65	×
	tsort	4142	2826	41	1275	0.01	85	×
	tty	3044	1995	16	1033	0.01	55	×
	uname	3216	2171	10	1035	0.0	55	×
	unexpand	3954	2864	43	1047	0.02	83	×
	uniq	4944	3745	30	1169	0.01	101	×
	unlink	3175	2041	31	1103	0.02	56	×
	uptime	5995	4748	61	1186	0.01	108	×
	users	3438	2268	43	1127	0.02	66	×
	vdir	18694	12845	329	5520	0.03	230	×
	wc	5172	3232	76	1864	0.02	89	×
	who	6421	5250	47	1124	0.01	93	×
	whoami	3167	2019	28	1120	0.01	58	×
	yes	3315	837	22	2456	0.03	34	×
Hopper	[	6654	5554	54	1046	0.01	96	
	b2sum	7513	5828	108	1577	0.02	103	
	base32	4370	3267	60	1043	0.02	83	
	base64	4309	3206	60	1043	0.02	83	

Hopper	basename	3250	2217	18	1015	0.01	59		
	basenc	5703	2628	50	3025	0.02	65		
	cat	3844	2734	59	1051	0.02	70		
	chcon	7993	5777	83	2133	0.01	147		
	chgrp	8453	6214	106	2133	0.02	150		
	chmod	8111	5919	97	2095	0.02	142		
	chown	8778	6548	93	2137	0.01	154		
	chroot	4375	3336	60	979	0.02	83		
	cksum	3385	2275	70	1040	0.03	70		
	comm	4252	3115	49	1088	0.02	86		
	cp	15287	11551	248	3488	0.02	244		
	csplit	19505	9033	153	10319	0.02	150		
	cut	4637	3677	27	933	0.01	91		
	date	14166	11978	509	1679	0.04	151	×	×
	dd	10307	7993	373	1941	0.05	132		
	df	12629	9468	224	2937	0.02	171		
	dir	18292	12607	350	5335	0.03	230	×	×
	dircolors	4122	3063	26	1033	0.01	86		
	dirname	3185	765	8	2412	0.01	28		
	du	28729	19356	564	8809	0.03	326		
	echo	3307	929	13	2365	0.01	34		
	env	4942	3870	43	1029	0.01	86		
	expand	3845	2742	111	992	0.04	79		
	expr	18927	16778	484	1665	0.03	246		
	factor	10210	8596	67	1547	0.01	152		
	false	2971	610	8	2353	0.01	23		
	fmt	4612	3686	29	897	0.01	91		
	fold	3720	2786	37	897	0.01	78		
	getlimits	4084	1600	36	2448	0.02	38		
	ginstall	18737	15318	296	3123	0.02	299		
	groups	3532	2455	23	1054	0.01	65		
	head	4989	3947	39	1003	0.01	82		
	hostid	3121	1987	30	1104	0.02	55		
	id	4572	3364	121	1087	0.04	76		
	join	5642	3123	135	2384	0.04	76		
	kill	3674	2613	40	1021	0.02	67		
	link	3177	2045	37	1095	0.02	57		
	ln	9018	5812	221	2985	0.04	154		
	logname	3141	2004	30	1107	0.01	56		
	ls	18292	12607	350	5335	0.03	230	×	×
	make-prime-list	594	406	52	136	0.13	18		
	md5sum	5461	4256	60	1145	0.01	88		
	mkdir	6688	5289	121	1278	0.02	107		
	mkfifo	3504	2389	62	1053	0.03	61		
	mknod	4022	2775	68	1179	0.02	65		

Hopper	mktemp	4797	3551	74	1172	0.02	107		
	mv	17717	14398	190	3129	0.01	294		
	nice	3573	2511	27	1035	0.01	58		
	nl	17870	15959	485	1426	0.03	227		
	nohup	3816	2529	92	1195	0.04	73		
	nproc	3595	2499	45	1051	0.02	66		
	numfmt	7444	6003	113	1328	0.02	94		
	od	8906	4379	145	4382	0.03	93		
	paste	3703	2766	19	918	0.01	71		
	pathchk	3373	2402	26	945	0.01	59		
	pinky	4134	1820	25	2289	0.01	68		
	pr	9771	8253	185	1333	0.02	158	×	×
	printenv	3082	718	8	2356	0.01	23		
	printf	6634	5426	63	1145	0.01	99		
	ptx	23639	20905	583	2151	0.03	265		
	pwd	3605	2573	26	1006	0.01	72		
	readlink	5243	2263	103	2877	0.05	59		
	realpath	5636	2644	86	2906	0.03	72		
	rm	8773	6288	99	2386	0.02	153		
	rmdir	5283	4164	74	1045	0.02	80		
	runcon	3127	768	8	2351	0.01	23		
	seq	6355	4836	100	1419	0.02	84		
	shred	7463	5746	151	1566	0.03	133		
	shuf	7400	5516	179	1705	0.03	135		
	sleep	3481	2325	45	1111	0.02	69		
	split	6754	5416	84	1254	0.02	117		
	stat	11306	9621	179	1506	0.02	173	×	×
	stdbuf	5986	4786	57	1143	0.01	96		
	stty	8443	7062	109	1272	0.02	101		
	sum	4789	2810	128	1851	0.05	80		
	sync	3354	2258	56	1040	0.02	60		
	tac	17883	15823	444	1616	0.03	228		
	tail	9384	5940	92	3352	0.02	117		
	tee	3828	2671	37	1120	0.01	78		
	test	6286	4976	35	1275	0.01	86		
	timeout	4157	2945	33	1179	0.01	93		
	touch	12189	10132	312	1745	0.03	140	×	×
	tr	5417	4129	69	1219	0.02	91		
	true	2972	610	8	2354	0.01	23		
	truncate	4023	3002	26	995	0.01	65		
	tsort	4109	2812	55	1242	0.02	85		
	tty	3033	1990	18	1025	0.01	54		
	uname	3203	2168	12	1023	0.01	54		
	unexpand	3924	2900	45	979	0.02	83		
	uniq	4906	3779	32	1095	0.01	102		

Hopper	unlink	3160	2049	35	1076	0.02	56		
	uptime	5968	4448	69	1451	0.02	93		
	users	3429	2269	43	1117	0.02	65		
	vdir	18292	12607	350	5335	0.03	230	×	×
	wc	5119	3243	92	1784	0.03	86		
	who	6353	5237	61	1055	0.01	92		
	whoami	3154	2016	30	1108	0.01	57		
	yes	3292	837	22	2433	0.03	34		
IDA Pro	[	6615	5554	54	1007	0.01	96		
	b2sum	7489	5828	108	1553	0.02	103		
	base32	4344	3267	60	1017	0.02	83		
	base64	4283	3206	60	1017	0.02	83		
	basename	3221	2217	18	986	0.01	59		
	basenc	5672	2628	50	2994	0.02	65		
	cat	3818	2734	59	1025	0.02	70		
	chcon	7967	5777	83	2107	0.01	147		
	chgrp	8426	6214	106	2106	0.02	150		
	chmod	8085	5919	97	2069	0.02	142		
	chown	8750	6548	93	2109	0.01	154		
	chroot	4346	3336	60	950	0.02	83		
	cksum	3358	2275	71	1012	0.03	70		
	comm	4227	3115	49	1063	0.02	86		
	cp	15254	11551	248	3455	0.02	244		
	csplit	19465	9033	131	10301	0.01	150		
	cut	4609	3677	27	905	0.01	91		
	date	14134	12122	423	1589	0.03	151		
	dd	10284	7993	373	1918	0.05	132		
	df	12592	9468	225	2899	0.02	171		
	dir	18220	12751	268	5201	0.02	230		
	dircolors	4092	3063	26	1003	0.01	86		
	dirname	3157	765	8	2384	0.01	28		
	du	28690	19356	564	8770	0.03	326		
	echo	3279	929	13	2337	0.01	34		
	env	4913	3870	43	1000	0.01	86		
	expand	3820	2742	111	967	0.04	79		
	expr	18891	16778	485	1628	0.03	246		
	factor	10182	8596	67	1519	0.01	152		
	false	2943	610	8	2325	0.01	23		
	fmt	4587	3686	29	872	0.01	91		
	fold	3693	2786	37	870	0.01	78		
	getlimits	4059	1600	36	2423	0.02	38		
	ginstall	18706	15318	296	3092	0.02	299		
	groups	3502	2455	19	1028	0.01	65		
	head	4962	3947	39	976	0.01	82		
	hostid	3093	1987	30	1076	0.02	55		

IDA Pro	id	4545	3364	121	1060	0.04	76
	join	5616	3123	135	2358	0.04	76
	kill	3647	2613	40	994	0.02	67
	link	3150	2045	37	1068	0.02	57
	ln	8989	5812	222	2955	0.04	154
	logname	3115	2004	30	1081	0.01	56
	ls	18220	12751	268	5201	0.02	230
	make-prime-list	593	406	52	135	0.13	18
	md5sum	5435	4256	60	1119	0.01	88
	mkdir	6661	5289	121	1251	0.02	107
	mkfifo	3476	2389	62	1025	0.03	61
	mknod	3995	2775	68	1152	0.02	65
	mktemp	4772	3551	74	1147	0.02	107
	mv	17681	14398	190	3093	0.01	294
	nice	3545	2511	27	1007	0.01	58
	nl	17832	15959	462	1411	0.03	227
	nohup	3789	2529	92	1168	0.04	73
	nproc	3568	2499	45	1024	0.02	66
	numfmt	7418	6003	113	1302	0.02	94
	od	8878	4379	145	4354	0.03	93
	paste	3675	2766	19	890	0.01	71
	pathchk	3346	2402	26	918	0.01	59
	pinky	4107	1820	25	2262	0.01	68
	pr	9744	8397	102	1245	0.01	158
	printenv	3053	718	8	2327	0.01	23
	printf	6607	5426	63	1118	0.01	99
	ptx	23596	20905	555	2136	0.03	265
	pwd	3573	2573	26	974	0.01	72
	readlink	5212	2263	105	2844	0.05	59
	realpath	5607	2644	87	2876	0.03	72
	rm	8742	6288	99	2355	0.02	153
	rmdir	5256	4164	74	1018	0.02	80
	runcon	3100	768	8	2324	0.01	23
	seq	6330	4836	100	1394	0.02	84
	shred	7438	5746	151	1541	0.03	133
	shuf	7379	5516	179	1684	0.03	135
	sleep	3452	2325	45	1082	0.02	69
	split	6731	5416	84	1231	0.02	117
	stat	11273	9765	96	1412	0.01	173
	stdbuf	5956	4786	57	1113	0.01	96
	stty	8419	7062	109	1248	0.02	101
	sum	4761	2810	128	1823	0.05	80
	sync	3328	2258	56	1014	0.02	60
	tac	17847	15823	444	1580	0.03	228
	tail	9363	5940	92	3331	0.02	117

IDA Pro	tee	3803	2671	37	1095	0.01	78
	test	6246	4976	35	1235	0.01	86
	timeout	4129	2945	33	1151	0.01	93
	touch	12160	10276	229	1655	0.02	140
	tr	5374	4129	69	1176	0.02	91
	true	2943	610	8	2325	0.01	23
	truncate	3995	3002	26	967	0.01	65
	tsort	4084	2812	55	1217	0.02	85
	tty	3005	1990	18	997	0.01	54
	uname	3176	2168	12	996	0.01	54
	unexpand	3899	2900	45	954	0.02	83
	uniq	4880	3779	32	1069	0.01	102
	unlink	3132	2049	35	1048	0.02	56
	uptime	5936	4448	69	1419	0.02	93
	users	3401	2269	43	1089	0.02	65
	vdir	18220	12751	268	5201	0.02	230
	wc	5091	3243	92	1756	0.03	86
	who	6327	5237	61	1029	0.01	92
	whoami	3126	2016	30	1080	0.01	57
	yes	3265	837	22	2406	0.03	34